PCT

WORLD INTELLECTUAL PROPERTY ORGANIZATION
International Bureau

INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(51) International Patent Classification 6 :
G06F 11/00, H04L

A2

(11) International Publication Number: WO 98/53399

(43) International Publication Date: 26 November 1998 (26.11.98)

(21) International Application Number: PCT/GB98/01367

(22) International Filing Date: 21 May 1998 (21.05.98)

(30) Priority Data:
97303472.1 21 May 1997 (21.05.97) EP
(34) Countries for which the regional or international application was filed: GB et al.

(71) Applicant *(for all designated States except US)*: BRITISH TELECOMMUNICATIONS PUBLIC LIMITED COMPANY [GB/GB]; 81 Newgate Street, London EC1A 7AJ (GB).

(72) Inventors; and

(75) Inventors/Applicants *(for US only)*: UTTON, Peter, Covington [GB/GB]; Tavern House, Station Road, Bentley, Ipswich IP9 2DB (GB). AKEHURST, David, Hesketh [GB/GB]; 28 Cherry Garden Road, Canterbury, Kent CT2 8EP (GB). SYMES, Andrew, John [GB/GB]; 110 Tennyson Avenue, Canterbury, Kent CT1 1ER (GB). JOHNSON, Adrian [GB/GB]; "Kelvindale", Heath Ride, Wokingham, Berkshire RG40 3QE (GB). BOULTON, David [GB/GB]; 45 Alfred Road, Kingston, Surrey KT1 2TZ (GB).

(74) Agent: DUTTON, Erica, Lindley, Graham; BT Group Legal Services, Intellectual Property Dept., Holborn Centre, 8th floor, 120 Holborn, London EC1N 2TE (GB).

(81) Designated States: AL, AM, AT, AU, AZ, BA, BB, BG, BR, BY, CA, CH, CN, CU, CZ, DE, DK, EE, ES, FI, GB, GE, GH, GM, GW, HU, ID, IL, IS, JP, KE, KG, KP, KR, KZ, LC, LK, LR, LS, LT, LU, LV, MD, MG, MK, MN, MW, MX, NO, NZ, PL, PT, RO, RU, SD, SE, SG, SI, SK, SL, TJ, TM, TR, TT, UA, UG, US, UZ, VN, YU, ZW, ARIPO patent (GH, GM, KE, LS, MW, SD, SZ, UG, ZW), Eurasian patent (AM, AZ, BY, KG, KZ, MD, RU, TJ, TM), European patent (AT, BE, CH, CY, DE, DK, ES, FI, FR, GB, GR, IE, IT, LU, MC, NL, PT, SE), OAPI patent (BF, BJ, CF, CG, CI, CM, GA, GN, ML, MR, NE, SN, TD, TG).

Published
Without international search report and to be republished upon receipt of that report.

(54) **Title:** OPERATIONAL ANALYSIS OF SOFTWARE-CONTROLLED SYSTEMS

(57) **Abstract**

The invention provides methods and apparatus for predicting and/or modelling performance characteristics of communications networks and information technology systems. These performance characteristics are generated from three separate models of the system, namely an application environment model of the application software. An execution environment model of the network and platforms on which the software executes and also a work load model for the system derived from anticipated user behaviour. These models are then combined together in a composite model data store so as to form a composite model of system behaviour. A performance model of the system is generated from the composite model. This Performance model allows performance characteristics of the modelled communications network or information technology system to be predicted. The invention further allows the three models to be generated from the actual software tools that are used to design the system. For example, where a Computer Aided Software Engineering (CASE) tool is used to develop the application software running on the system, the invention allows the output of the same CASE tool (400) to be used as input to generate the application model of this application software. Similarly, the output (405) of the network design tool used to design the physical and/or logical structure of the network underlying the application software may be used to generate the execution environment model of the network behaviour. System scenarios may also be generated. These System Scenarios allow different disigns of the system to be modelled. For example, the system may be modelled as if the application software had differing configurations in its distribution across the execution environment.

FOR THE PURPOSES OF INFORMATION ONLY

Codes used to identify States party to the PCT on the front pages of pamphlets publishing international applications under the PCT.

AL	Albania	ES	Spain	LS	Lesotho	SI	Slovenia
AM	Armenia	FI	Finland	LT	Lithuania	SK	Slovakia
AT	Austria	FR	France	LU	Luxembourg	SN	Senegal
AU	Australia	GA	Gabon	LV	Latvia	SZ	Swaziland
AZ	Azerbaijan	GB	United Kingdom	MC	Monaco	TD	Chad
BA	Bosnia and Herzegovina	GE	Georgia	MD	Republic of Moldova	TG	Togo
BB	Barbados	GH	Ghana	MG	Madagascar	TJ	Tajikistan
BE	Belgium	GN	Guinea	MK	The former Yugoslav Republic of Macedonia	TM	Turkmenistan
BF	Burkina Faso	GR	Greece			TR	Turkey
BG	Bulgaria	HU	Hungary	ML	Mali	TT	Trinidad and Tobago
BJ	Benin	IE	Ireland	MN	Mongolia	UA	Ukraine
BR	Brazil	IL	Israel	MR	Mauritania	UG	Uganda
BY	Belarus	IS	Iceland	MW	Malawi	US	United States of America
CA	Canada	IT	Italy	MX	Mexico	UZ	Uzbekistan
CF	Central African Republic	JP	Japan	NE	Niger	VN	Viet Nam
CG	Congo	KE	Kenya	NL	Netherlands	YU	Yugoslavia
CH	Switzerland	KG	Kyrgyzstan	NO	Norway	ZW	Zimbabwe
CI	Côte d'Ivoire	KP	Democratic People's Republic of Korea	NZ	New Zealand		
CM	Cameroon			PL	Poland		
CN	China	KR	Republic of Korea	PT	Portugal		
CU	Cuba	KZ	Kazakstan	RO	Romania		
CZ	Czech Republic	LC	Saint Lucia	RU	Russian Federation		
DE	Germany	LI	Liechtenstein	SD	Sudan		
DK	Denmark	LK	Sri Lanka	SE	Sweden		
EE	Estonia	LR	Liberia	SG	Singapore		

OPERATIONAL ANALYSIS OF SOFTWARE-CONTROLLED SYSTEMS

This invention lies in the field of apparatus for, and methods of, operational analysis of software-controlled systems and finds particular application in apparatus for, and methods of, performance analysis of systems such as those used in information technology and communications.

The development of communications technology such as:

- Asynchronous Transfer Mode (ATM);
- Synchronous Digital Hierarchy (SDH); and
- digital subscriber lines (xDSL)

has brought about increasing demand for information technology such as broadband and multimedia systems, including:

- video-on-demand and video conferencing systems;
- Internet and Intranet systems; and
- other communications systems such as telemarketing call centres.

Such systems are complex to design, build and modify. They typically consist of complex software application program(s) residing on hardware platform(s) distributed over one or more communications networks. These networks are often physically distributed across a wide area such as a city or a country.

At present, effective testing of such systems is only possible after installation, at which point it is often difficult and costly to overcome failings in system design.

Typically, such failings arise in relation to the performance of the system rather than the functionality of the system. For instance, failure to meet response time specifications, failure of the system to be scaleable, bottle necks within networks etc. Such failings in the performance of a system are often the result of unforeseen interactions between modules of the software application program(s) and the hardware platform(s) of the system. Other failings in system performance, particularly in communications systems, arise from the response of the system to surge loading and shock loading by system operators.

Accordingly it would be preferable for developers to be able to take performance into account in the design phase of a system as part of the development process. "Evaluation of software quality attributes during software

design" by C Wohlin, Informatica 18 (1994), 55-70, is a paper describing a method for predicting performance characteristics of a communications system during development phases of the system life cycle.

To do this Wohlin develops a performance model of the system. The performance model consists of three interacting sub-models. These are the software model, the architecture model and the usage model.

Wohlin's software model is a program representing the performance characteristics of the application programs used within the communications system.

Wohlin's architecture model is a program representing the performance characteristics of the hardware platforms and network configurations used within the communications system.

Wohlin's usage model is a program representing user behaviour and interaction with the system. It is this behaviour and interaction with the communications system that generates system loading characteristics.

Wohlin combines these three model components to emulate the real world interaction of the application program(s), hardware platform(s), network configuration(s) and user behaviour of the system being developed. Accordingly, the linked models operate to predict the performance characteristics of the communications system as a whole.

This approach requires that performance models be generated at the outset of the design life cycle. These performance models need to be generated in a common language so that they can be linked together.

Currently these models cannot be made using conventional system development tools and so Wohlin's approach requires personnel with expertise in the field of system modelling and simulation.

Further, Wohlin's approach does not readily form a design aid for the development of communications systems. Generally, a design aid will allow a system designer to experiment with different system designs and layouts.However, Wohlin requires modifications to each of the component models in order for such experimentation to take place. This again requires the use of personnel with expertise in the field of system modelling and simulation.

To reduce the need for expert personnel, a design aid would ideally utilise system design information generated by system design tools. For example:

1) operational information defined within software applications; and

2) topological information of a system's architecture.

However, these two separate forms of design information cannot be simply combined into a performance model.

Each set of information is generated by a separate tool, for example software applications may be designed using Computer Aided Software Engineering (CASE) tools and a system's architecture may be designed using a network design tool.

However software applications are commonly designed so that they can operate independently of the network on which they run. For example, a client \ server software application generally contains a single definition of a client and a single definition of a server.

However in an operational system the client software and the server software may each be installed several times over on several different hardware platforms.

Hence, it is typical for the software application to contain little, if any, information on the topology of the network on which it is, or is to be, installed.

Similarly, networks are designed to operate independently of the software applications that are running on them. Accordingly there is little support within network design tools for incorporating detailed information about all of the various modules within the software applications that each platform within the network may need to support.

Hence, neither the software application nor the network design contains sufficient detail for the interactions between the modules of the software application and the hardware platforms of the network to be predicted.

However, a performance model requires information on these interactions. Accordingly design information generated by software application and network design tools cannot be simply transformed into a common format and combined into a performance model.

1. STATEMENTS OF INVENTION

According to a first aspect of the present invention there is provided a method of generating performance models of information technology or communications systems comprising the steps of:

1) inputting to modelling apparatus at least a first data set of operational information of said system;

2) inputting to said modelling apparatus at least a second data set of topological information of said system's architecture;

3) transforming said first and second data sets into at least one performance model of said system.

Preferably this method provides that at least a first data set comprises a software application and wherein said method further comprises the step of:

4) extracting said operational information from said software application.

Preferably this method further provides that at least a second data set comprises a definition of said system's architecture and wherein said method further comprises the step of:

5) extracting said topological information from said definition.

This has the advantage that disparate and / or incompatible data sets, such as those generated by Computer Aided Software Engineering tools and network architecture design tools can form input to a process and / or method that then automatically generates a performance model. This also allows performance modelling to incorporated into the system's development cycle.

Preferably the method further comprises the steps of:

6) extracting at least a first set of interface data from said first and second data sets;

7) linking said first set of interface data with a predefined interface whereby operatively associating said first and second data sets.

This interface in an system designed according to object oriented methodologies may be an Object Request Broker (ORB). The use of an ORB allows interaction between the two input data sets, namely operational information derived from a software application and topological information derived from an architectural design.

Preferably step 6) further comprises the steps of:

8) identifying inter-related sub-sets of data within said operational information;

9) deriving at least part of said interface data from said identified sub-sets.

In a system designed using object oriented design methodology this step is a process of detecting used classes.

According to a second aspect of the present invention, there is provided system performance simulation apparatus for use in analysing performance in an information technology or communications system, the apparatus comprising:

i) a data store for storing at least a first model of a first aspect of the system and at least a second model of a second aspect of the system;

ii) means for extracting or generating, and storing, linkage information for linking the first and second models; and

iii) transformation means for transforming said first and second models, using said linkage information, into a performance model of the system.

Preferably, the apparatus further comprises performance testing means for performance testing the system, using said performance model.

Preferably, the apparatus further comprises result loading means for returning performance test results to the data store.

Preferably, the data store is adapted to store a third model of a third aspect of the system. The three models may then comprise an applications model, an execution environment model and a workload model. These three model types are clearly important in the construction of a performance model of the overall system.

The models of the various aspects of the system may be generated by any appropriate modelling tool, or tools, and these tools need not necessarily generate models having a common format. The system may therefore also be provided with means for ensuring that the storage of the models in the data store is done according to a common format, to facilitate the transformation to a performance model. This may be done by providing one or more download mechanism(s) for downloading the output of said one or more modelling tools, the download mechanism(s) providing a conversion function for the output of any modelling tool not already conformant to said common format.

It is preferable that the performance results returned to the data store are accessible to the modelling tool used to generate each model so that a new or modified model could be generated which takes the performance results into account. Where different modelling tools are used to generate different respective models, it is also preferable that the performance results are available to each

modelling tool. This is supported by embodiments of the present invention in which a view of the performance results which relates to the source models can be provided by linking the results, in the data store, to source constructs.

In this specification, the initials "CMDS" are used. These stand for "Composite Model Data Structure", which is the data structure stored in the data store. The CMDS provides in a single data structure a composite model of a system whose performance is to be analysed. That single data structure may comprise for instance the applications model, environment model and workload model mentioned above, together with linkage information relating all three models to each other.

Practical benefits of the above CMDS format are:

- it integrates aspects of a system by linking together component models into one composite model which can then form the basis for performance modelling. CMDS enables the operational performance aspects of a system to be modelled and evaluated while maintaining the association with the other aspects of the system
- by providing an independent representation, it enables interfaces to other tools to be developed
- it allows the composite model to be checked for integrity and completeness
- it allows performance results to be reported in the context of the different aspects of the system.

Preferably, the single data structure further comprises routing information in respect of components of the environment model. Further, preferably, the linkage information comprises routing or address information for components of the applications model with respect to components of the environment model as well as routing information for components within the environment model alone.

Embodiments of the present invention are preferably based on an object-oriented approach to software development modelling. Object oriented software technology is now widely established and is described for instance in the book entitled "Object-Oriented Analysis and Design" by Grady Booch, published in 1994 by Benjamin/Cummings.

An object-oriented ("OO") approach to software development models a system as a number of objects which interact. The basic engineering building blocks of an OO system are classes and objects. An object comprises data

together with methods for operating on the data. A class defines methods and data attributes common to a set of objects of similar type. An object is an instance of a class.

Embodiments of the present invention involve in particular "Unified Modelling Language" (UML), an industry-standard language for specifying, visualising, constructing and documenting the artifacts of software systems. Development of UML has been led by Rational Software Corporation and information about UML can be obtained for instance via the Worldwide Web at www.rational.com/uml. In particular, the CMDS preferably comprises UML constructs, and thus can be used to generate a performance model which reflects the structure of an object-oriented application.

The performance model may be embodied for example in Information Processing Graph (IPG) notation. This is a known representation for "Queuing Network Models" (QNM) which can be solved using "Discrete Event Simulation" (DES) and analytic tools, referred to for instance in the book "Performance Engineering of Software Systems" by Connie Smith, published in 1990 by Addison Wesley. The use of IPG notation allows a solution tool (performance testing means) such as the SES Workbench tool to be used, this tool being further identified later in this specification. The approach of the transformation process can then include the generation of a QNM of the system, represented in IPG notation. A file is created, using the SES query language, to represent the system model. This file can then be input to the SES Workbench tool (the performance testing means) to generate a model for solution.

In principle, the specific SES Workbench tool can be replaced by any general purpose DES tool since it should be possible to derive a suitable simulation model from an IPG description. For instance, it could be replaced by a Petri-Net based simulator.

According to a second aspect of the present invention, there is provided a method of generating a performance model of an information technology or communications system, which method comprises the steps of:

i) storing in a data store at least a first model of a first aspect of the system and at least a second model of a second aspect of the system;

ii) extracting or generating, and storing in the data store, linkage information for linking the first and second models;

iii) transforming said first and second models, using said linkage information, into a performance model of the system.

Preferably, at least one of the models comprises an interaction (or "scenario") diagram to trace, in the system, the execution of a scenario in time for instance as a sequence of messages between objects. The method may then further comprise performance testing the system, using the performance model, and the modification of the first or second model, and/or of the interaction diagram and repetition of steps i) to iii) above.

This modification and rerunning of the performance testing, from the design environment through to the performance testing environment, is a particularly beneficial capability of embodiments of the present invention. It can enable design engineers to take performance results into account without having to have, or to contract in, the different skills of the performance engineer.

A performance analyser for use in designing and modifying software-controlled systems, according to an embodiment of the present invention, will now be described, by way of example only, with reference to the accompanying drawings in which:

Figure 1 is a representation of a hardware platform and network configuration of an information technology system of the type to which a performance analyser of the present invention may be applied;

Figure 2 is a representation of of the software application programs which reside on the hardware platforms of Figure 1;

Figure 3 is a high level view of the performance analyser architecture in it's three functional sections;

Figure 4 is a files and conversion tools view of the architecture of Figure 3;

Figure 5 is schematic view of the hardware environment in which the performance analyser sits;

Figures 6 to 8 show the operations happening at each of the three sections of Figure 3;

Figures 9 to 13 show example outputs of the source design tools Rose and Configurator;

Figure 14 shows a model parameters file for use in converting a CMDS data structure into the SES environment;

Figures 15 to 22 show aspects of a generated performance model in the SES environment;

Figure 23 shows a user interface for viewing the performance analyser from the source to the SES environments;

Figure 24 an example CMDS data structure once populated;

Figures 25 to 31 show CMDS classes and objects and their behaviours from the source environment;

Figure 32 shows a state chart from the source environment;

Figure 33 shows a CBN example;

Figure 34 shows a system scenario;

Figures 35 and 36 show translation mechanisms for translating Rose and Configurator files;

Figures 37 to 45 show various aspects from the SES environment;

Figures 46 to 49 show examples from worked scenarios;

Figure 50 outlines the CMDS to SES translation process;

Figure 51 shows an object interaction;

Figures 52 to 73 show the aspects set out in their respective titles; and

Figure 74 shows a default ORB submodel.

BACKGROUND TO PERFORMANCE ANALYSIS

Before commencing with a detailed description of the performance analysers, it is advantageous to provide further background to the art of performance analysis.

An information technology system for which it may be desirable to analyse and predict its performance characteristics is now described, with reference to Figures 1 and 2. The system concerned is an Access Operations Unit (AOU) that manages a local access network of a Public Switched Telephone Network (PSTN) (not shown).

The hardware environment of the AOU is shown in Figure 1. This hardware environment, known as an "execution environment" (EE), comprises hardware platforms 100-130 and their interconnection via a local area network 140 and a wide area network 135.

The AOU is a client/server arrangement from an operator's point of view. It consists of a number of clients 110 communicating with one or more server

115. The servers 115 are in communication with databases 120, customer support systems 125, works management systems 130, switching management centres (not shown in Figure 1) and at least one access network analysis server 100.

The software modules which run in the execution environment of Figure 1 are shown in Figure 2. These constitute applications and can be referred to as the application layer. They are distributed across the hardware platforms of the execution environment. These software modules interact. It is these interactions which define the functional behaviour of the system.

For the purpose of performance analysis the interactions between the software modules may be considered as consisting primarily of two categories of behaviour: 1) the transmission of data from one software module to another; and 2) the processing of this data.

When data is transmitted from one software module of the AOU to another it will typically be passed across either the WAN 135, the LAN 140 or both (unless the software module to which the data is passed is on the same hardware platform as the software module transmitting the data).

In operation, the performance of the AOU will be affected by workload. For instance, the speed of the AOU will depend partly on how much data is being passed across the network. If there is more data than the network can efficiently cope with, then the speed of the system will be adversely affected.

Also, the speed of the AOU will depend partly on the speed with which the data can be processed. If the data cannot be processed fast enough then the speed of the AOU will be adversely affected. The speed of the processing is determined, at least in part, by both the volume of the data to be processed and the speed of the processor that is processing the data.

A performance model estimates performance characteristics such as the speed and response times with which a system, such as the AOU of Figures 1&2, will operate. To do this, a performance model will typically simulate the amount of data that is generated by the system, where this data flows to within the system and how long the system takes to process it. This can be achieved using a statistical technique known as Discrete Event Simulation (DES). Tools supporting DES techniques are often adapted to receive, as input, performance models known

as Queued Network Models (QNM). The DES tool executes the QNM and generates statistical data relating to the QNM.

A performance analyser according to an embodiment of the present invention automatically generates a QNM representation of a system being analysed. The QNMs are automatically created from at least three sets of input data, which respectively relate to the application software, the execution environment on which the application software is installed and the anticipated workload of the system.

The input data to the performance analysers may be derived directly from system design tools such as software design CASE tools, and network design tools. This allows actual system design information to be fed directly into the performance analyser. Hence the performance analysers can be seen to derive performance models directly from design information with limited, if any, assistance from performance engineering experts.

ARCHITECTURE OF THE PERFORMANCE ANALYSER

Referring to Figure 3, the performance analyser consists of three sections: Section 1 - input of data from the source tools; Section 2 - a Composite Model Data Structure (CMDS) and operations on it; and Section 3 - a Performance Results Generator.

The first section consists of four parts, one for input of data in relation to each of four different aspects of a system. These aspects are a software application 305, an execution environment 310, a work load specification 315 and a system scenario 320.

The tool used for source input of each of the first three types of components above is Rational Rose version 4.0, using the UML notation, and for the System Scenario tool, BT Configurator version 1.2 which is described in co-pending patent application GB 9707550.1, and incorporated herein by reference. The BT configurator tool is similar to the Rose tool but has different lexical rules and grammar. These together with a library are given later in this specification, under the heading "Configurator Grammar".

These tools are not essential to embodiments of the present invention. Alternative, commercially available, tools include Software Through Pictures (STP) by Aonix, San Francisco, CA, USA for software modelling, and Network Draw Plus

by Network World, Framingham, Massachusetts, USA; or Visual Thought by Confluent Inc, San Francisco, CA, USA for Hardware platform and Network configuration modelling.

Input of the source data to the CMDS is achieved via a number of conversion or translation programs. The translation process takes the data files saved by the tool used to generate the source data as input, generates a database representation of it, and maps elements of the source data to elements in the CMDS. Hence, elements of the CMDS are created from elements of the source data.

There is a database used to support the CMDS, usually referred to herein as the "Permabase" database, and which is also used as an intermediate step in the translation process from the source tools to the CMDS. Commercially available object oriented data bases and tools such as O2 and Versant may be used to provide the Permabase functionality.

The modelling tool used is SES Workbench version 3.1.4. It is necessary to translate the CMDS 300 to a structure for use in the modelling tool environment and this is done by mapping information from the CMDS into a performance model by using the SES query language.

Figure 4 shows a files and conversion tools view of the overall architecture of the performance analyser. (The files are shown by two parallel lines, the conversion tools by rounded corner boxes). Files 400, 405, 410, 415 output by the source tools are converted and loaded as a CMDS file 420. Loading involves populating the CMDS data structure 300 by means of four conversion operations:

- Detect Used Classes 330
- Flatten and Multiply 335
- Generate Routing Tables 340
- Interaction Diagrams to CBN 325

The information in the CMDS data structure is then mapped into a performance model 425 which can be supported by the SES workbench environment. This is done by creating query language files 430 which can be used by a query interpreter to generate the SES model 425.

Referring to Figure 5, the performance analyser may itself be supported in a distributed system, for instance communicating by means of a Local Area

Network (LAN) 500. As shown, the design tools 530, 535 are loaded on a server 505 which is local to a user's client workstation 510. Data storage 515 is available over the LAN 500 and the performance modelling environment 345 is also remote from both the client workstation 510 and the server 505 carrying the design tools 530, 535.

However, the data stores and functionality can be loaded wherever there is sufficient capacity and acceptable access. Alternatively, the performance analyser might be supported at a single location with tools, performance modelling environment 520 and data storage 515 all residing on the user's local server 505.

Referring to Figures 6, 7 and 8, the three sections of Figure 3 can be expanded as follows.

Referring to Figure 6, the design engineer works within the Rose tool to generate design files 600, 605, 610. The design engineer works in the Configurator tool to create system scenario file outputs 615.

Looking at generation of the application components file 610, this is a normal use of the Rose design tool. The Rose design tool is intended for modelling software applications and specifies an application in terms of application objects and the relationships between them. The objects and relationships are determined by the design engineer. Figures 9 and 10 show examples of a class diagram and an interaction diagram created within the Rose tool by the design engineer for the particular example of retrieving a page from a server. Referring to Figure 9, the class diagram shows the object classes that need to be present, in this case client, server, page address and page classes. The client and server classes have an interaction indicated. Referring to Figure 10, the interaction diagram shows a sequence of interactions in chronological order, for the "retrieve page" example, triggered by an external event 1000, between the classes of the class diagram of Figure 9.

Generating the output files for the execution environment 600 and for the workload 605 is a less usual use of the Rose design tool. It is necessary for the CMDS data structure 300 to be populated with descriptions of workload and execution environment components in order to be able to derive a performance model. This is done using functionality of the Rose design tool, which can equally be applied to non-software components.

Referring to Figure 11, a class diagram for the execution environment output file 600 shows platform components defined by platform type and attributes. Similarly, referring to Figure 12, the workload output file 605 comprises a class diagram of relevant workload components.

These platform and workload components are usually fairly static and can be defined in a library.

The examples in Figure 12 show an open thinker 1200 and a closed thinker 1205 workload components. Although these show the same attributes, they have different names, these different names determining that different behaviour will be relevant. This becomes apparent when the library components represented by the workload component names are pulled in. The library component definitions will show different behaviour for an open thinker from a closed thinker.

In practice, it is the system scenario which pulls together application, workload and execution environment components. The system scenario relates the components in a representation of the whole system, for loading in the CMDS data structure. Referring to Figure 13, the system scenario diagrams developed within the Configurator tool 535 are therefore important since they relate the components from the workload and execution environment to objects of the application layer. In Figure 13, one workload component 1300 can be seen. In the page request (or "cache") example described above, this workload component relates to an object in the class "client" from the application source model 610. Since this is a specific instance of the object class client, it is shown as an object 1305 and given an individual object name "cl1". The object 1305 from the application source model 610 relates to a platform component, in this case from the class "PC". This is shown in the system scenario diagram as a platform instance 1310 and given a unique platform name "PC1". In running an example of the cache exercise, further execution environment components have to be specified. These are two network instances 1315, 1320 and a platform instance 1325 of the class "SUN" (this being a specific type of server). Lastly, a second application object, this time of the class "server" 1330, relates to the SUN execution environment component 1325.

Notably, it is only objects from the application class diagram which will interact. That is, although for instance there will be execution environment

instances of the classes "client" and "server" in a system scenario, it is only the software component equivalents, from the application class diagram, which will have a sequential interaction and thus appear in an interaction diagram.

Referring to Figure 7, the output files 600, 605, 610, 615 from the source tools Rose and Configurator 530, 535 are converted to files in a PERMABASE database format 565, 570, 575, 585, 580 prior to populating the CMDS data structure 300. The data structure 300 provides a kind of interface between the output of the Rose and Configurator tools and the input to the SES performance environment. To do that, it needs to hold the output of the Rose and Configurator tools in a way which can be fed to the SES environment and the Permabase database format provides an intermediate conversion.

The CMDS data structure 300 provides a framework for information which has to be populated with the data contained in the output of the Rose and Configurator tools. Figure 7 shows four stages in the population process, "CMDS1" through to "CMDS4".

The first areas of the CMDS data structure 300 to be populated are class lists, interactions written in Common Behaviour Notation (CBN), and linked object structures representing the system scenarios from Configurator. The main conversion step on first loading data to the data structure is therefore step 1 as shown, the conversion of the interaction diagrams to CBN.

To progress towards having extracted sufficient data from the Permabase input to feed a performance model, there are then three additional important steps:

- Detect Used Classes (Step 2 as shown)
- Flatten and Multiply (Step 3 as shown)
- Generate Routing Table (Step 4 as shown)

The first of these reflects the fact that the performance model needs to be able to model the performance of all the classes which may be involved in relation to a platform component, during running of a software process. It would be possible to model all classes in relation to all platform components but that would make the performance model unwieldy. Alternatively, it would be possible to look at the scenario diagrams and only model the classes which are shown there to be involved in relation to each platform component. However, that would not take account of all classes. For instance, it will not take account of classes which arise

during running of the software, called herein "dynamic objects", such as pages which are delivered from a remote site.

The solution in embodiments of the present invention is to pull in data from the interaction diagrams and to use that data to search recursively the behaviour of the classes which arise in relation to a platform. The "Detect Used Classes" step therefore increases the number of classes in the linked object structure to accommodate for instance the creation of dynamic objects, and classes which "nested" within other classes.

Step 3 reflects the fact that the performance model needs every class instance to be taken into account. The output of the Configurator tool allows sub-models to be hidden within modules. It also allows multiple instances to be represented just once. Step 3 pulls out every instance and gives it a unique identifier, for instance related to the platform it resides on and, if there are multiples, a number in a sequence.

Step 4 is important in that the performance model needs to be able to track routes between platform components, in order to predict traffic effects for instance. The generation of a routing table is done from the system scenarios, simply looking at the routes available between platform components.

Once the Steps 1 to 4 have been carried out, every part of the CMDS data structure has been populated ready for loading to the SES environment. Referring to Figure 8, it is now necessary to convert the information in the data structure 300 to Queued Network Models (QNMs) which the performance environment is able to execute. This is done using SES Query Language in order to get the appropriate conversion.

Referring to Figure 14, there are some minor parameters which the user needs to input and this is done by means of the model parameters file shown. The parameters which have to be set are such things as warmup time, runtime, and whether collection statistics are required for relevant for the classes relevant to the model to be run. Figure 14 shows a model parameters file for the cache example.

Referring again to Figure 8, the result of the conversion is a Meta Model 800 and a library of component definitions 805. These together provide a generated performance model.

Referring to Figure 15, it is possible to view the meta model 800 at different levels, including at the catalogue level. This provides a directory of

modules in the meta model 800 which can be defined from the library of definitions 805. Each module can be "drilled into" to view the sub-modules from which it is constructed.

The catalogue level introduces another of the inventive aspects of the present invention, the ORBware module 1500. Although, as described previously, a routing table is constructed in the CMDS data structure 300 which gives connectivity between platform components, in order to run a performance model it is necessary also to know where the application components are in relation to each other. This is done by the introduction of an ORB (Object Request Broker) layer. Each platform component has associated with it an Object Adapter (OA) which sits between the platform component and the application layer. The OA holds information about all the application components located at its platform component. If an incoming transaction arises for one of it's application components, the OA can send it to the correct one. If an outgoing transaction arises from one of it's application components, intended for an application component on a remote platform, the OA can supply the information as to what platform the target application component is sitting on. The routing table information can be used to reach the remote platform and then the OA at the remote platform determines which of it's application components should act on the transaction.

Figures 16 to 22 show examples of levels at which the meta model for a generated performance model for the cache example can be viewed by the user.

Figure 16 shows submodels of each of the execution environment instances. It is possible to view each submodel in turn and the user can access a Queued Network Representation (QNM) for each one present.

Figure 17 shows submodels from the workload layer and Figure 18 shows one of these expanded into it's QNM. The submodel information as expanded, for execution environment components and workload environment components, can be pulled in from a library created using the SES Workbench environment.

Figure 19 shows an application module and Figure 20 shows the server submodel. This latter shows the application behaviour for the server class, compiled from the source models. The notation is that known for showing a system running, in particular the paths between components and behaviours. Transactions (units of work) arise and follow arcs from icon to icon which indicate

a sequence of behaviour for dealing with the transaction. Behavioural icons might for instance represent queuing, blocking queues, waiting and branches.

Figure 21 shows system scenario modules for the cache example and Figure 22 shows part of the linkages between the different layers for the "cache1" submodel. In essence, Figure 22 shows delays etc for transactions flowing through topology arcs between objects from all the environments.

Referring to Figure 23, embodiments of the present invention can present a very powerful user interface by means of which performance models can be run, source models modified and the performance model rerun. The four buttons to the left of Figure 23 2300 allow the user to view each of the source models by starting up the relevant design tool (Rose or Configurator). The source models can be modified as necessary and loaded to the CMDS data structure 300. The CMDS button 2305 allows the user to view interim files and these can be loaded to the SES environment. The "model" button 2310 connects the user to UNIX and generates an SES model in the Workbench environment.

Referring to Figure 24, an example of activation of the CMDS button 2305 starts a browser and gives the user the structure of the system as stored in the CMDS data structure 300.

The following parts of the description of this specification expand on the operations described in overview above.

OVERVIEW OF INPUT MODELS

(It should be noted that the headings in the following section start at a heading numbered "2.3". This simply reflects the structure of the document from which the section was taken and it should not be understood that there is descriptive material missing.)

2.3 Requirements for a Performance model

The information required in a UML application design for a performance model to be generated is:

A specification of the components which make up the application; and specification of how those components behave and interact with each other.

There are two things which affect performance within a system, the amount of data, and the time taken to process that data. The amount of data mainly affects the time taken to transmit it around the system, and the time taken to process it is mainly dependant upon the speed of the processor doing the processing.

The basic building block of an application designed using object oriented techniques is an Object. Applications are collections of communicating objects – the communications being the passing of objects back and forth between other objects of the system. Objects are a composition of related data and functionality (methods), the specification of which is contained in a Class specification. All Objects are instances of a Class, the behaviour – functionality – remains the same, but with differing data values.

To generate a performance model from this collection of objects it is necessary to know the following two pieces of information about each class, it can be given directly or calculated from its parts:

- The amount of data contained within the class,
- The time taken to execute its methods.

When it comes to Distributed Systems, or at least systems containing multiple threads, performance is also affected by the interaction between the threads. This impacts the UML source specification such that it is preferable to specify how each class of object will handle multiple threads – is the object single threaded and threads are queued up until the object is free, or can it handle multiple threads at the same time, and if so how many.

2.3.1 Data Size

The data size of, or Space used by a class can be given as one of its properties, or calculated from the size of its attributes. This size could be used when passing

(transmitting) objects of this type between other objects, or to keep track of a memory resource on a Platform.

2.3.2 Method Execution Time

The execution time of a method can be given as one of its properties, or calculated from its behaviour. The actual time taken will depend upon the platform the object is executing on, so the execution time needs to be given in platform independent units, then the platform can specify how many of these units it can process per second. Given that this is the case, the actual units used makes no difference provided there is consistency between the object and the platform as the units will cancel out of the equation. However, the choice of units does depend on the ability of the Application designer to estimate the quantity for the method to execute in, and the ability to specify how many per second a platform can execute.

2.3.3 Classes and threads

The UML as it stands allows the specification of various (designer defined) properties on a class, these can be used to indicate the thread management style of that class.

2.4 Class Behaviour Specifications

In any particular system model there will be some classes which are assumed to be the *primitive* classes – the building blocks on which all other classes are based. At an early stage in the project there could be quite high level classes, which would obviously be subdivided at a later stage, but are currently considered primitive. Alternatively a primitive class can be one which represents a portion of the system which is not to be modelled but nevertheless has functionality used by the modelled part, with a predictable (or previously predicted/modelled/worked out) execution time. In terms of performance modelling, these primitive classes have no

behaviour description beyond a specification of the *time complexity* of their methods.

Non-primitive classes have their behaviour described in terms of other classes (which eventually resolve to primitive ones), or rather calls to methods on objects belonging to the other classes. The methods may also include a time complexity value indicating some additional work done by the method, that is not included in the behaviour description.

Class behaviour, in the UML, can be specified in three ways:

- Interaction diagrams
- Statecharts
- Documentation

In terms of performance modelling, it is preferable that there be consistency between these three. This may be achieved using compatible defined semantics for each method. A notation – referred to as common behaviour notation which is effectively a form of pseudo code which is compatible with both Interaction diagrams and Statecharts - is detailed below under the heading "CBN Generation".

2.4.1 Dynamic Object Creation

In a single Platform environment dynamic creation of an object can be equated to calling a method on the class which creates the required object. In a multi-platform (distributed) environment, there is the question of which platform should the new object be created upon. The general idea behind Distributed Object Programming is to hide the underlying execution environment, and the decision of where the new object should reside is handled externally to the application. The performance analyser uses a simple rule that "All objects are created locally".

Probability Tests

During the high level design stages, it is often known that a choice of path will be made, but the detail of how that choice will be made are not yet decided. There is still the requirement to generate a performance model at this stage. This may be

achieved by a Probability test – the clause in a Condition Option may be a Probability test as opposed to a Boolean expression.

3. Execution Environment Specification

The Execution Environment is the collection of physical components that the (or a number of) application(s) executes upon. This consists primarily of Platforms – on which the processing is carried out, and Networks – which allow the platforms to communicate with each other.

In-between the Objects of the Application and the Platform of the EE there is a notional ORB layer. This layer consists of an OA (Object Adapter) residing at each platform, and a Name Server. The ORB provides location transparency to each of the objects within the system – all method calls to another object go via the ORB. The Name Server informs the ORBs where to forward method calls to, in order that they reach the target object.

A Platform consists of: a processor for executing the work units specified for object methods at a specific rate; a connection to one or more networks; and an OA (and/or link to the ORB layer).

A Network may be as simple as a straight forward link between platforms, or it may consist of a number of links connected via bridges, switches, platforms etc.

If we treat these two basic components as classes, we can determine what needs to be specified for each.

Platform

Attributes:

speed – The speed of the processor, rate at which work units are executed.

Methods:

transmit – Send a method call to an object via a connected network.

process – Execute a number of work units on this platform.

Network

Attributes:

speed	-- Rate at which data can be transmitted.
delay	-- delay incurred by data transmitted over this network.

Methods:

transmit	-- Send data (method call) across this network.

This provides a simple interface to any component that these two interact with.

4. Workload Specification

The workload layer consists of the definition of "classes" of user behaviour. The performance analyser provides two user models, an open and a closed thinker model. Both models cyclically think and activate a method on an object, the closed model waits for a response from the method before continuing with the cycle (equivalent to a synchronous method call), where as the open thinker model does not (equivalent to an asynchronous method call). Both models take as parameters a think time, and a distribution type for that time.

5. The System to Model

The 3 layers, Workload, Application, and Execution Environment define components which can be used to build a system. To get a Performance model of a system, it is necessary to have a definition of the system itself, not just a definition of its components.

The UML does have some mechanisms which address the allocation of Application components to Platforms (or Processors), but does not address the whole system, leaving out: the definition of how those Platforms are connected; the initial state of the system (in order for an application to execute there need to be objects, not just classes); and how the application is to be driven.

The System Scenario definition from the configurator tool defines the instances of the components that exist to make up the system, and defines how those components are connected together.

A system contains:

- Workload Instances driving application components
  - Behaviour, when, how, which functionality to invoke
- Objects communicating over and residing on an EE
  - Behaviour, result of method invocation – other objects invoked
  - Data
- Execution Environment Component Instances
  - Platform Instances
    - OA (ORB link) for handling object communication and existence
    - Processor(s) for executing behaviour
    - Network connections for communication to other platforms
  - Networks Instances
    - links between (multiple) platforms

Component Models of the Architecture

The CMDS

The CMDS is the core of the performance analyser tool set, it provides a complete model of the system, containing all the information from each source tool, combined into a consistent model.

The CMDS is a data model built on an object oriented database. The CMDS data model is defined in the UML diagrams of Figures 25 to 34.

Rose To Intermediate Database

The files saved by Rose (mdl files) are well structured and are suited to storage in an intermediate database. It consists of a number of nested (what it calls) 'objects', each of which can be converted to an entity in an intermediate database in a straightforward one to one mapping.

The conversion process uses a *bison++* and *flex++* defined grammar (decendants of *yacc* and *lex* that are c++ oriented, ***"Flex++ a C++ extension to the lexical analysers Flex and Lex M.E. "Lesk and E. Schmidt. Lex - A Lexical Generator. Technical report, Rational Software Corporation, 1995"***, and ***"Bison++ a C++ extension to the parsers Bison detailed in the reports: "Charles Donnelly and Richard Stallman. Bison the YACC-compatible Parser Generator. Technical report, Free Software Foundation, November 1995" and "Yacc Stephen C. Johnson. YACC: Yet Another Compiler-Compiler. technical report, Bell Laboratories, July 1978"***, which constructs an intermediate database out of the Parsed 'objects', their attributes, and attribute values. The two grammar files are listed below under the headings "Rose Grammar" and "Configurator Grammar". Figure _____ the components of the conversion tool.

Configurator To Intermediate Database

This tool is very similar to the Rose to intermediate database translation tool, except it uses a different set of lexecal and grammar rules – see below _____. The configurator file (dwg file) is also divided into sub units which map well to Entities, Attributes, and Attribute values. Figure shows the components of this tool.

Components To CMDS

This tool takes the intermediate database representations of the Rose input and maps the Rose entities to CMDS entities, or to be more exact, CMDS entities are created from the Rose entities.

Many entities in Rose map to their duplicates in the CMDS, this is because the component representation in the CMDS is based on the UML OO notation, which is supported (in part) by Rose.

The following sections define which Rose entities are used to create the CMDS entities. Although the application design will have been entered using the UML notation in Rose, these conversions are necessary because: Rose does not support all of the UML notation; Rose stores the UML design using a lot of the Booch notation attributes and entities; and the CMDS UML meta model may not be exactly like the Rose one in other ways.

Dependency

name = name
documentation = documentation
supplier = supplier
export_control = exportControl

Refinement_Relationship

name = name
documentation = documentation
steriotype = steriotype

Generalization

name = name
documentation = documentation
steriotype = steriotype
supplier = supplier
export_control = exportControl
friend = friend
virtual = virtual
discriminator = *from documentation*
powertype = *from documentation*
constraints = *from documentation*

Class_Attribute from Association

determine which is the part end and which is the aggregate end

name = name of part end

type = supplier of part end

containment = containment in properties of part end.

Class_Attribute

name = name

documentation = documentation

type = type

initial_value = initv

is_static = static

is_derived = derived

containment = 'By_Reference' if type ends in '&', else 'By_Value'.

export_control = exportControl

Parameter

name = name

type = type

initial_value = initv

style = 'By_Reference' if type ends in '&', else 'By_Value'.

Resource_Consumer from a string in the documentation attribute of an Operation

Parse the string, and get an Entity with attributes being the components of the string

if no error during parsing

name = name

type = type

command = command

quantity = quantity

frequency = frequency

limit_type = limit_type

limit = limit

Operation

name = name
documentation = documentation
visibility = exportControl
return_type = new Operation_Result(result)
behaviour = new Attribute_Set_Att
properties = new Attribute_Set_Att
properties[concurrency] = concurrency
properties[time_complexity] = time_complexity
properties[exceptions] = exceptions
properties[protocol] = protocol
properties[space] = space
parameters list = parameters list, with items converted to Parameters
consumers = new Attribute_List_Att;
for each line in documentation that contains "%RC"
add to consumers list, a new Resource_Consumer created from the rest of the line

Operation_Result (String type)
class = type without the '&'.
style = 'By_Reference' if type ends in '&', else 'By_Value'.

Role

name = name
documentation = documentation
supplier = supplier
properties = new Attribute_Set_Att
properties[multiplicity] = cardinality
properties[constaints] = constaints
properties[containment] = Containment
properties[export_control] = exportControl
properties[is_friend] = is_friend
properties[is_static] = is_static

properties[is_aggregate] = is_aggregate
properties[is_navigable] = is_navigable
properties[is_principle] = is_principle
qualifiers list = keys list with items converted to Parameter

Event
name = name
parameters = parameters

Action_Time
when = when
condition = condition
event = new Event created from Event

State_Action
name = name
action_time = new Action_Time created from ActionTime

Send_Event
target = target
action_time = new Action_Time created from ActionTime
event = new Event created from Event

State_Transition
action = action
condition = condition
supplier = supplier
send_event = new Send_Event created from sendEvent
event = new Event created from Event

State

name = name

type = type

actions list = Action's in actions list converted to State_Action's
+ SendEvent's in actions list converted to Send_Event's

transitions list = transitions list, with items converted to State_Transition

statechart = new Statechart created from statemachine

Statechart

states list = states list, with items converted to State

Class

name = name

documentation = documentation

stereotype = stereotype

abstract = abstract

cardinality = cardinality

concurrency = concurrency

export_control = exportControl

module = module

persistence = persistence

space = space

relationships = new Attribute_List_Att; //Filled in elsewhere.

dependencies list = used_nodes list, with items converted to Dependancy

generalizations list = list of items (converted to Generalization) in superclasses list with no stereotype

refinement = item converted to Refinement_Relationship, in superclasses list with stereotype = refines

attributes list = class_attributes list with items converted to Class_Attribute

operations list = operations list with items converted to Operation

behaviour = new Attribute_Set_Att

behaviour[statechart] = new Statechart created from state_machine

Association

name = name

documentation = documentation

stereotype = stereotype

derived = derived

constraints = Constraints

roles list = roles list, with items converted to Role

Check if Association is a valid Attribute

find one of its roles with is_aggregate = true

Find the Aggregate Class of an association

find one of its roles with is_aggregate = true

aggregate class is the supplier of this role

Object_Link

supplier = supplier

association = Association

client_stereotype = client_visibility

supplier_stereotype = supplier_visibility

messages list = messages list, with items converted to Message

Object

name = name

class = class

documentation = documentation

is_muli = multi

persistence = persistence

collaborators list = collaborators list, with items converted to Object_Link

Message

name = name

documentation = documentation

synchronization = synchronization

direction = dir
Parse the name String, to get an entity with attributes being the components of a message spec.
if no error in parsing
guard = guard
sequence_expression = sequence_expression
recurrence = recurrence
result_name = result_name
method = method
parameters = parameters

Scenario (Interaction Specification)
objects list = logical_models list, with items converted to Object

Package
name = name
documentation = documentation
stereotype = stereotype
dependencies list = visible_categories list with items converted to Dependency
scenarios = list of Mechanism's (converted to Scenario) in logical_models list
contents = list of Class_Category's (converted to Package) in logical_models_list
+ list of Class's (converted to Class) in logical_models list
for each Association in logical_models list
convert it to Association
if it is a valid Attribute
add to the aggregate classes attribute list the Association converted to a Class_Attribute

System Scenario To CMDS

This tool takes the intermediate database representation of the Configurator Input, and maps the Configurator entities to CMDS entities, likewise – to the components to CMDS tool – the CMDS entities are created from the Configurator ones. For the

definition of the basic library components used in the performance analyser, see

_____.

SystemScenarioToCMDS

For each configurator Symbol
Convert it to a CMDS entity.
Add it to the System Scenario components.
Make Connections.
Connect Plugs to Sockets.

Make Connections

For each configurator Connection
Find the CMDS entity equivalent of the Symbol at each end of the connection.
Connect the two entities.

Connect Plugs To Sockets

Connect Plugs to Sockets for each subsystem within this one.
For each Access_Point_Plug in this subsystem
Find the Access_Point_Socket with the same name as the plug in the subsystem it is connected to,
Add the plug connections to the Socket external connections.
For each item externally connected
replace the connection to the plug with a connection to the socket.

Configurator Symbol to CMDS Entity mappings

The following sections define which Configurator entities are used to create the CMDS entities.

Workload_Instance

Created from a **Symbol** with *SymbolName* containing "WORKLOAD_INSTANCE".

class = pointer to the "Class" entity with name = attributes[WORKLOAD_CLASS]

design_parameters = attributes[DESIGN_PARAMETERS]

name = attributes[WORKLOAD_NAME]

object = pointer to the App_Object paired with this object via a **Connection** entity.

App_Object

Created from a **Symbol** with *SymbolName* containing "APP_OBJECT".

class = pointer to the "Class" entity with name = attributes[OBJECT_CLASS]

design_parameters = attributes[DESIGN_PARAMETERS]

constructor_parameters = attributes[MODEL_PARAMETERS]

name = attributes[OBJECT_NAME]

platform = pointer to the Platform_Instance paired with this object via a **Connection** entity.

workload = pointer to the Workload_Instance paired with this object via a **Connection** entity.

Platform_Instance

Created from a **Symbol** with *SymbolName* containing "PLATFORM_INSTANCE".

class = pointer to the "Class" entity with name = attributes[PLATFORM_CLASS]

connections = list of pointers to Network_Instances paired with this object via a **Connection** entity.

design_parameters = attributes[DESIGN_PARAMETERS]

name = attributes[PLATFORM_NAME]

objects = list of pointers to App_Objects paired with this object via a **Connection** entity.

Network_Instance

Created from a **Symbol** with *SymbolName* containing "NETWORK_INSTANCE".

class = pointer to the "Class" entity with name = attributes[NETWORK_CLASS]

connections = list of pointers to Platform_Instances or Network_Instances paired with this object via a **Connection** entity.

design_parameters = attributes[DESIGN_PARAMETERS]

name = attributes[NETWORK_NAME]

SubSystem

Created from a **Symbol** with *SymbolName* containing "SUBSYSTEM".

name = attributes[SUBSYSTEM_NAME]

number_of_instances = attributes[NUMBER_OF_INSTANCES]

access_points = List of Access_Point_Sockets within this subsystem.

components = SystemScenarioToCMDS(attributes[LINK][entities])

Access_Point

Created from a **Symbol** with *SymbolName* containing "ACCESS_POINT_SOCKET" or "ACCESS_POINT_PLUG".

name = attributes[ACCESS_POINT_NAME]

internal_connections = list of pointers to instances inside the subsystem.

external_connections = list of pointers to instances outside the subsystem.

Detect Used Classes

This algorithm gives each platform instance a pointer to the definition of each class that is used on it. The algorithm recursively searches the behaviour of each class on the platform to find the classes used within those classes, until all have been

detected. This process is started by the list of classes of the objects residing on the platform as defined in the System Scenario.

A class is considered used by a platform, if at any time an object of that class resides on the platform. This will occur if:

a) An object is dynamically created.
b) An object is returned 'By Value'.
c) An object is passed as a method parameter 'By Value'.

Detect Used Classes

For each platform
For each object on the platform
Add the classes of these objects to the used classes list,
Get the classes used by these classes,
Add those to the list of used classes also.

Get Used Classes

For each operation defined on the class
If a parameter is passed to it by value, add the parameter class to the used classes list
For each method called within the operation
If it is a constructor call, add the class constructed to the used classes list
If it returns a result by value, add the result class to the used classes list
Get the classes used by the ones just detected,
Add these to the used classes list.

Flatten and Multiply

This algorithm flattens out the hierarchy in the System Scenario and duplicates the multiple instances, so that each instance in the CMDS System Scenario representation contains a unique instance for every one defined.

The hierarchy relates to the manner in which the configurator defines system scenarios. It permits a system scenario to be defined as a series of submodels, each of which may contain further submodels.It is these submodels within submodels that provide the hierarchy that is flattened by this process. Similarly, the submodels need to be multiplied up.

The instance names have to be unique, this is achieved by extending the names by adding the name of the subsystem they are within when flattened out, and by adding a number when duplicating multiple instances.

Flatten and Multiply

Fix Parameter Names
Multiply Subsystems
Realign Entity Pointers
Remove Access Points
Expand Subsystems
Multiply Platform Instances
Multiply App Instances

Fix Parameter Names

For each parameter instantiation within the system scenario
Convert the value from the user definition convention into the system convention.
{i.e. turn xxx[n].yyy[m] into xxx_n_yyy_m}

Multiply Subsystems

For each subsystem within this one
Multiply subsytems
repeat 'number of instances'-1 times
duplicate this subsystem
realign duplicates entity pointers
add new subsystem to this ones components
realign original subsystems entity pointers

Realign Entity Pointers

For each subsystem within this one
realign its entity pointers
For each component within this subsystem
For each Entity pointer within the component
create a new pointer to the entity in this subsystem with the same name as original entity pointed to.

Remove Access Points

For each subsystem
remove access points of subsystem
For each Access Point of this subsystem
For each internal connection
add to the internal component, connections to each external component
remove the connection to this access point from the internal component
For each external connection
add to the external component, connections to each internal component
remove the connection to this access point from the external component

Expand Subsystems

For each subsystem within this one
expand its subsystems
extend its component names with its name and instance number
refine any of its component parameters referring to it as a specific subsystem instance
add its components to the components of this subsystem
remove subsystem from this ones components

Multiply Platform Instances

For each platform
repeat 'number of instances'-1 times
duplicate platform
add instance number to new platform name

create duplicate objects and workloads equivalent to those attached to original platform
attach them to new platform
add instance number of platform to object and workload names
add all new objects to the system scenario
add connections to new platform to networks platform is connected to
add instance number to original platform
add instance number of original platform to original object and workload names

Multiply App Instances
For each object
repeat 'number of instances'-1 times
duplicate object
add instance number to new object name
create duplicate workloads equivalent to ones attached to original object
attach new object to platform
add instance number of object to workload names
add instance number to original object and to original workloads

CBN Generation

The Common Behaviour Notation (CBN) referred to above is a pseudo code like notation which describes the behaviour of a class when a method is called on it. The behaviour of a Class is entered in either an Interaction Diagram or Statechart as part of the Application component input. The performance analyser generates CBN from an Interaction Diagram, although conversion from Statecharts may also be implemented.

The following algorithm describes the Interaction to CBN conversion.

Create a list of messages associated with each operation in each Class definition, from the interaction specifications.

Convert the message list for each operation into CBN.

Optimise the CBN.

Create message list

For each Interaction Specification

messList = list of all Messages in the interaction, adding to each message references to client and supplier of the message.

For each Object in the Interaction Spec.

From the messList

Add a list of all messages sent to this object.

Add a list of all messages sent from this object.

Convert Guard numbers into operation names. (must be done here, numbers are local to each interaction spec.)

For each message sent to this object

Using the sequence_expression, add to the operation spec. in the class def of this object, a list of messages from this object which are as a result of this message.

Convert messages to CBN

For each operation

Create a Sequential Composition for the operation.

For each list of messages

Create Sequential Composition from list

Add this to the statements of the operation Seq.

Optimise the CBN

Remove immediately nested Seqs.

Combine sequential duplicate conditionals.

Combine sequential conditionals if they refer to the same Probability variable.

Creation of Sequential from message list

Order the messages.

For each message

If new Index number

If sequence number contains a letter
Create a Parallel Composition, add it to sequential statements.
Convert message to statement.
Add it to Parallel statements.
Else
Convert message to statement.
Add it to Sequential statements.
Else
If sequence number contains a letter
Convert message to statement.
Add it to previous Parallel Statements.
Else
Convert message to Condition Option.
Add it to previous Condition Statement.

Convert Message To Statement

If sequence expression contains a recurrence part
If first char is a '*'
create Iteration Statement from message
Else
create Condition statement from message
Else
If there is a guard
create a Guarded statement
Else
create a method call

Method Call

Created from a *Message* (with client and supplier pointers added).

call_style = synchronization
result_name = result_name
target = supplier[name] (or client if message is sent the other way, or 'THIS' if message is sent to itself.)

operation = operation

parameters = parameters

Condition Statement

Create Condition Option from message

Add it to the list of options.

Condition Option

clause = recurrence (without the '[', ']')

statement = Method Call created from message.

Iteration Statement

clause = recurrence (without the '*','[', ']')

statement = Method Call created from message.

Guard

operations = list of operations in message guard list.

Remove the guard list from the message.

Statement = message converted to a statement.

Routing Table Generation

The routing table simply generates from the system scenario information a set of all the shortest routes between each combination of Platform pairs. A route consists of a *from* and *to* platform name, and a number of network name *steps*. The search algorithm is a breadth first search.

CMDS to SES

This conversion process is described in greater detail below.

However, an overview of this process shows that a QNM model of the system being analysed is generated in a text-based format. This QNM forms input to a DES tool.

The performance analyser uses the SES workbench detailed above. Accordingly, the performance analyser generates a QNM using the SES query language.

SES to CMDS

The SES model collects the following results:

i) time to execute any method call made,
ii) processing time taken by a method on a platform
iii) transmission time of a message + data across a network
iv) utilisation of platforms
v) utilisation of networks

Each of these results is a set of statistics. The first three sets contain: mean; standard deviation; minimum; maximum; variance; sample size, where as the utilization sets contain: period in use; number of periods; shortes period; longest period; utilization (period in use / simulation time).

All results are stored as though they were method calls made by one object on another, the result is stored in the calling object, and specifies the target object, the method called, and the statistic set. Utilization, processing times and transmission times are stored as method calls by the platform or network on themselves. Calls made by dynamically created objects, will cause a representation of the object to be added to the System Scenario, connected to the relevant platform.

Rose Grammar

Lexical Rules

```
"-"?[0-9]+              { return mdlParser::INTEGER;}

"-"?[0-9]+\.[0-9]+      { return mdlParser::FLOAT;}

\"[^"]*                 { if (yytext[yyleng-1] == '\\') {
                        yymore();
                       } else {
                        char c = input();
                        if (c != '"')
                          cerr << "Scanner Error in Parsing String." << endl;
                        yytext[yyleng] = '"';
                        yytext[yyleng+1] = '\0';
                        return mdlParser::STRING;
                       }
                      }

\@[0-9]+                { return mdlParser::OBJECT_LABEL; }

\("-"?[0-9]+\," ""-"?[0-9]+\)       { return mdlParser::INT_TUPLE; }

\(                      { return mdlParser::L_PAR; }
\)                      { return mdlParser::R_PAR; }

"|"[^\n]*\n             { theLine++; return mdlParser::LABEL_TEXT_LINE; }

"\n"                    { theLine++; }
" "                     |
"\t"                    ;

.                   {fprintf( stderr,"Unrecognised character: %s\n", yytext);}
```

```
object                  return mdlParser::OBJECT;
list                    return mdlParser::LIST;
TRUE                    return mdlParser::TRUE_TOK;
FALSE                   return mdlParser::FALSE_TOK;
value                   return mdlParser::VALUE;
Text                    return mdlParser::TEXT;
cardinality             return mdlParser::CARDINALITY;

[A-Z,a-z,_]+            return mdlParser::NAME;
```

Grammar Rules

```
%token L_PAR R_PAR
%token STRING INTEGER FLOAT BOOLEAN INT_TUPLE OBJECT_LABEL
%token LIST VALUE LABEL_TEXT_LINE TRUE_TOK FALSE_TOK TEXT
                                                    CARDINALITY
%token OBJECT NAME

MDLfile
 : object_petal
   object_design
 ;

object_petal
 : object
     {
       if (Integer_Att($$["version"]) != Integer(40)) {
         cout << "Mdl file version : " << Integer_Att($$["version"]) << endl;
         cout << "This parser is for mdl files version : 40." << endl;
         exit(1);
       }
     }
```

```
    ;

  object_design
  : object {mdl_design = $1;}
  ;

  object
  : L_PAR OBJECT object_details R_PAR
    {
      $$ = $3;
    }
  ;

  object_details
  : object_type object_id object_label attribute_list
    {
      $$ = new Entity_Att(String_Att($1));
      copyAttributes(ToAttribute_Set_Att($4),ToAttribute_Set_Att($$));
      ToAttribute_Set_Att($$).Attribute_Set_Att::operator[]("name") = $2;
    }
  ;

  object_label
  : /* empty */
  | OBJECT_LABEL
  ;

  object_id
  : /* empty */
    {
      $$ = new String_Att();
    }
  | string object_id
```

```
    {
      $$ = $1;
    }
   ;

attribute_list
   : /* empty */
     { $$ = new Attribute_Set_Att(); }
   | attribute_name attribute_value attribute_list
     {
         $$ = $3;
       $$[String_Att($1)] = $2;
     }
   ;

attribute_value
   : object
   | list
   | set              { $$ = new String_Att("** a set");}
   | value
   | string
   | boolean
   | INTEGER          { $$ = new Integer_Att(atoi(yylloc.text)); }
   | FLOAT            { $$ = new Real_Att(atof(yylloc.text)); }
   | OBJECT_LABEL
   | INT_TUPLE        { $$ = new String_Att("** a tuple");}
   ;

list
   : L_PAR LIST list_type list_contents R_PAR
     { $$ = $4; }
   ;

list_contents
```

```
    : object_list
    | string_list
    | tuple_list
    ;

object_list
    : /* empty */
      {
          $$ = new Attribute_List_Att;
      }
    | object object_list
      {
          $$ = $2;
        if (!$1.unassigned()) {
          ToAttribute_List_Att($$).Add($1);
          }
      }
    ;

string_list
    : string
    | string string_list
    ;

string
    : STRING
      {
        String s = String_Att($1).at(1,String_Att($1).length()-2);
        s.gsub("\\\"","\"");
        $$ = String_Att(s);
      }
    | label_text
    ;
```

```
tuple_list
 : INT_TUPLE
 | INT_TUPLE tuple_list
 ;

set
 : L_PAR string INTEGER R_PAR
 ;

value
 : L_PAR VALUE value_type string R_PAR { $$ = new String_Att($4); }
 ;

label_text
 : LABEL_TEXT_LINE
   { $$ = new String_Att( String_Att(String_Att($1).after("|")).before("\n") ); }
 | LABEL_TEXT_LINE label_text
   {
      $$ = new String_Att( String_Att($1).after("|") + String_Att($2) );
   }
 ;

boolean
 : TRUE_TOK      { $$ = new Boolean_Att(true); }
 | FALSE_TOK     { $$ = new Boolean_Att(); }
 ;

value_type
 : TEXT
 | CARDINALITY
 ;

object_type
```

```
    : NAME
    ;

  list_type
    : /* blank */
    | NAME
    ;

  attribute_name
    : NAME
    | VALUE
    ;
```

Configurator Grammar

Lexical Rules

```
"-"?[0-9]+                  { return dwgParser::INTEGER;}
"-"?[0-9]+\.[0-9]+          { return dwgParser::FLOAT;}
\"[^"]*                     { if (yytext[yyleng-1] == '\\') {
                              yymore();
                            } else {
                              char c = input();
                              if (c != '"')
                                cerr << "Scanner Error in Parsing String." << endl;
                              yytext[yyleng] = '"';
                              yytext[yyleng+1] = '\0';
                              return dwgParser::STRING;
                            }
                           }
\("-"?[0-9]+\." ""-"?[0-9]+\)   { return dwgParser::INT_TUPLE; }

\(                          { return dwgParser::L_PAR; }
\)                          { return dwgParser::R_PAR; }
":"                         { return dwgParser::COLON; }
\\                          { return dwgParser::BACKSLASH; }

"\n"                        { theLine++; }
" "                         |
"\t"                        ;

T                           return dwgParser::TRUE_TOK;
F                           return dwgParser::FALSE_TOK;
";;;"[^\n]*                 return dwgParser::COMMENT;

NIL                         return dwgParser::NIL;
```

```
LINES                   return dwgParser::VALUE;

Annotation              return dwgParser::ENTITY;
Symbol                  return dwgParser::ENTITY;
Connection              return dwgParser::ENTITY;

[A-Za-z_][A-Za-z0-9_\-]*    return dwgParser::NAME;


.                       {fprintf( stderr,"Unrecognised character: %s\n", yytext);}
```

Grammar Rules

```
%token L_PAR R_PAR COLON BACKSLASH
%token STRING INTEGER FLOAT BOOLEAN INT_TUPLE
%token TRUE_TOK FALSE_TOK
%token ENTITY NAME COMMENT NIL VALUE

%%
DWGfile
  : COMMENT
    L_PAR
    contents
    R_PAR
    COMMENT
      { dwg_design = $3; }
  ;

contents
  : /* empty */
    { $$ = new Attribute_Set_Att;
      $$["entities"] = new Attribute_List_Att;
    }
```

```
    | attribute contents
      {
          $$ = $2;
          copyAttributes(ToAttribute_Set_Att($1),ToAttribute_Set_Att($$));
      }
    | entity contents
      {
          $$ = $2;
        ToAttribute_List_Att($$["entities"]).Add($1);
      }
    ;

  attribute
    : L_PAR NAME value R_PAR
      {
          if (String_Att($2) == String("LINK")) {
            $$ = new Attribute_Set_Att;
            $$[String_Att($2)] = convertFileToPbDb(String_Att($3));
          } else {
            $$ = new Attribute_Set_Att;
            $$[String_Att($2)] = $3;
          }
      }
    ;

  entity
    : L_PAR ENTITY contents R_PAR
      {
          $$ = new Entity_Att(String_Att($2));
          copyAttributes(ToAttribute_Set_Att($3),ToAttribute_Set_Att($$));
      }
    ;

  value
```

```
    : /* empty */        { $$ = Attribute_Ref(); }
    | entity
    | string
    | boolean
    | INTEGER            { $$ = new Integer_Att(atoi(yylloc.text)); }
    | FLOAT              { $$ = new Real_Att(atof(yylloc.text)); }
    | int_list
    | name_list
    | string_list
    | NIL                { $$ = Attribute_Ref(); }
    | attribute_list
    | style
    | VALUE
    | value_set
    ;

attribute_list
    : L_PAR attributes R_PAR
      { $$ = $2; }
    ;

attributes
    : attribute
      {
          $$ = new Attribute_Set_Att;
          copyAttributes(ToAttribute_Set_Att($1),ToAttribute_Set_Att($$));
      }
    | attribute attributes
      {
        $$ = $2;
          copyAttributes(ToAttribute_Set_Att($1),ToAttribute_Set_Att($$));
      }
    ;
```

```
style
  : COLON style_name
  ;

style_name
  : NAME
  | NAME BACKSLASH style_name
  ;

string_list
  : L_PAR strings R_PAR
    { $$ = $2; }
  ;

strings
  : string
  | string strings
  ;

string
  : STRING
    {
      String s = String_Att($1).at(1,String_Att($1).length()-2);
      s.gsub("\\\"","\"");
      $$ = String_Att(s);
    }
  ;

boolean
  : TRUE_TOK        { $$ = new Boolean_Att(true); }
  | FALSE_TOK       { $$ = new Boolean_Att(); }
  ;

int_list
```

```
    : L_PAR integers R_PAR
      { $$ = $2; }
    ;

integers
    : INTEGER
      {
          $$ = new Attribute_List_Att;
          ToAttribute_List_Att($$).Add($1);
      }
    | INTEGER integers
      {
          $$ = $2;
          ToAttribute_List_Att($$).Add($1);
      }
    ;

name_list
    : L_PAR names R_PAR
      { $$ = $2; }
    ;

names
    : NAME
      {
          $$ = new Attribute_List_Att;
          ToAttribute_List_Att($$).Add($1);
      }
    | NAME names
      {
          $$ = $2;
          ToAttribute_List_Att($$).Add($1);
      }
    ;
```

```
value_set
 : L_PAR
   value_set_entries
 R_PAR
 ;

value_set_entries
 : value_set_entry
 | value_set_entry value_set_entries
 ;

value_set_entry
 : INTEGER
 | COLON style_name
 | NIL
 | value_set
 ;
```

System Scenario Library Components

Workload_Instance

Workload_Name - The 'Unique' (within this subsystem) name of the Workload Object.

Workload_Class - The Class of the Workload Object.

Parameters - Values to set in the Workload Object.

Documentation - User notes.

Object

Object_Name - The 'Unique' (within this subsystem) name of the Object.

Object_Class - The Class of the Object.

Number_Of_Instances - The number of instances of this object the symbol is to represent.

Design_Parameters - Values to be set in the Object.

Constructor_Parameters - Parameters to be passed to the constructor of this object when created.

Documentation - User notes.

Platform_Instance

Platform_Name - The 'Unique' (within this subsystem) name of the Platform Object.

Platform_Class - The Class of the Platform Object.

Number_Of_Instances - The number of instances of this platform and objects on it the symbol is to represent.

Parameters - Values to set in the Platform Object.

Documentation - User notes.

Network_Instance

Network_Name - The 'Unique' (within this subsystem) name of the Network Object.

Network_Class - The Class of the Network Object.

Network_Class - The Class of the Network Object.

Parameters - Values to set in the Network Object.

Documentation - User notes.

SubSystem

SubSystem_Name - The 'Unique' (within this subsystem) name of the Subsystem.

Number_Of_Instances - The number of instances of this subsystem the symbol is to represent..

Documentation - User notes.

Access_Point_Plug

Access_Point_Name - The 'Unique' (within this subsystem) name of the Access Point Plug.

Documentation - User notes.

Access_Point_Socket

Access_Point_Name - The 'Unique' (within this subsystem) name of the Access Point Socket.

Documentation - User notes.

Common Behaviour Notation

The *Common Behaviour Notation* referred to above will now be described. CBN captures the common constructs of the differing UML notations for interaction diagrams and state charts. It then describes the mapping from each UML behaviour notation into this common form. This could be viewed as specifying some tighter semantics on the UML notation, and Breu et al (Ruth Breu et al; Towards a Formalization of the Unified Modeling Language; http://www4.informatik.tu-muenchen.de; Institut fur Informatik Technische Univrtsitat Munchen) addresses some of these issues.

The syntax of the CBN is specified in a form akin to BNF (Backus – Naur Form). The specification of a syntax plays two roles: one, to aid the specification of the construct; and two, to give a method of writing the behaviour specifications.

There are two parts to this notation, which are consistent with the commonly understood concept of Object Oriented (or Based) Software design.

Firstly, behaviour is captured in classes, that is to say, things which exhibit behaviour are objects of a particular class, and that class describes the objects behaviour. Behaviour occurs as a result of stimulation (possibly self stimulation in the case of an *active* object - one with its own thread of control), and this stimulation is via method calls. Methods (or Operations) are defined for each object, and the behaviour of the object is defined within these methods.

Secondly, the description of the behaviour is made up of how objects interact, or more specifically, what, for a given method call on an object, other methods are called on what other objects and in what order. The basic action here is a method call, but we also need constructs to bind various method calls together.

Class

Attributes

A class consists of a number of methods and attributes, methods define the behaviour, and attributes contain the 'state', of the object. An attribute is a

relationship with another object, these can be either, *By Reference* or *By Containment*.

A reference to another object gives this object a handle on it, so that this object can call methods on the referenced one. Initially the references will point nowhere, and need to be assigned to an object before use.

A contained object, is one which has its lifetime tied to that of its parent, i.e. when the parent gets created, so do all of its children, there is no need to assign them separately.

Syntax

The syntax used by the performance analyser is:

```
class : 'CLASS' class_name [':' inheritance_list] '{' [attribute]* [method]* '}' ;

class_name : String ;

inheritance_list
  : String                    { A class name. }
  | String ',' inheritance_list    { A list of class names. }
  ;

attribute : class_name ['&'] attribute_name ; {The '&' indicates 'By Reference'.}

method : [return_class] method_name '(' [parameter_defs] ')' cbn_action ;

attribute_name : String ;

return_class : class_name ['&'] ; {The type of the object returned, '&' indicates a
                                   reference.}

method_name : String ;
```

```
parameter_defs
  : class_name ['&'] param_name          {The '&' indicates pass by reference.}
  | class_name ['&'] param_name ',' parameter_def
  ;
```

Basic Behaviour Actions

Looking at the UML Interaction diagrams (which will be the prime source of behaviour definition) and various programming, and formal method languages, the following constructs may be regarded as a basic set:

- Sequential Composition
- Parallel Composition
- Conditional Statement
- Iteration Statement
- Guarded Statement

The following subsections describe the semantics of the method call and constructs, and gives a possible textual syntax for them.

Syntax

```
cbn_action          { A Common Behaviour Notation action is a, }
  : method_call             { method call or }
  | sequential      { sequential construct or }
  | parallel        { parallel construct or }
  | conditional             { conditional statement }
  | iteration       { iteration statement or }
  | guard                   { guard statement. }
  ;
```

Note on syntax:

```
    [ xxx ]             { xxx is optional. }
```

[xxx]+ { xxx may be repeated 1 or more times. }
[xxx]* { xxx may be repeated 0 or more times. }
{ xxx } { xxx is a comment. }

Method Call

A method call can be made Asynchronously, or Synchronously. Asynchronous calls cause a new thread to be created, the current thread *fires and forgets* the method call then continues with its own behaviour, the called method generates a new thread in the target object. Synchronous calls keep the same thread taking it with the call and bringing it back with the return.

Asynchronous

An Asynchronous method call does not return a result, methods which do return a result can be called asynchronously, but the result is not sent back to the calling object. An Asynchronous method call does not make use of the Exception catching clauses.

Synchronous

There are different forms of Synchronous method call. The basic form is for the method to be called, if the object cannot (or will not) accept the call – i.e. its busy – then the thread is queued and blocked indefinitely, until the method is accepted. This can of course lead to deadlock, and gives a reason for having alternatives. Two such alternatives are, Balking and Timeout. A Timeout method call, carries an extra *time* parameter indicating the length of time it stays queued until accepted, if the time is exceeded the call is rejected, and an exception is raised in the calling method. A balking method call is a special case of a Timeout, with time set to 0, if the object cannot accept the call immediately, it is rejected.

Assignment

There are two forms of assignment: one which creates a copy of the object; and one which just creates a new name for the object. In most programming languages assignment is treated as a construct in its own right, but it can be seen (both forms) as a method call.

e.g.

```
x := y
  →
x := y.this()
```

or `x := y.clone()`

y.this() returns the object refered to by the name y (as does 'this' inside a classes methods) and gives it an additional name x; and y.clone() returns a copy of y (taken from the idea in Java) and gives it the (first) name x.

Dynamic Construction

It is frequently necessary to create new objects as part of the behaviour, this can still be seen as a method call, but a call to the *class* rather than to a specific object. The call returns an object of the type of the class, after having executed the behaviour defined in the appropriate constructor. The constructor is behaviour that occurs as a result of the object being created, there can be more than one constructor, each taking a different parameter list. To destroy an object we can call a method 'delete' on the object.

e.g.

```
x := ClassX.new()
x.delete()
```

Return

A method may return a result, in order for this to happen the method behaviour must specify the object (result) to be returned. This return statement can be seen

an asynchronous method call, the target being the calling object of this method, and the result passed as a parameter.

Syntax

The above sections describe a number of different types of method call: the basic synchronous and asynchronous styles for user defined methods; and "special" methods for handling specific behaviour – this(), clone(), new(), delete(), return(). They can all be defined by the same syntax:

```
method_call :
  call_style [result_name ":="] target "." method "("[parameters]")" ["CATCH"
exception cbn_action]*
;

call_style
  :     "ASYNC"   {Asynchronous.}
  |     "SYNC"    {Synchronous.}
  |     ""        {Assignment.}
  |     ""        {Construction.}
  |     ""        {Destruction.}
  |     ""        {Return.}
;

result_name : String ;    {A name used to refer to the object returned as a result
                          of the method call.}

target : String ;         {The name of the object to call the method on.}

method : String ;  {The name of the method to call.}

parameters
  : String        {The name of an object to pass as a parameter (or a const
                  value).}
```

| String "," parameters {A sequence of "," separated names of objects (or const values).}

;

exception : String ; {The name of an exception to catch.}

statement : {A Common behaviour notation construct or method call.}

Sequential Composition

The contained actions occur sequentially in the order listed.

Syntax

sequential : "SEQ" "{" [cbn_action]* "}" ;

Parallel Composition

The contained actions occur in parallel, and this statement ends when all contained actions have ended.

Syntax

parallel : "PAR" "{" [cbn_action]* "}" ;

Conditional Statement

A condition statement consists of a number of options, the first option with a clause that equates to True is the path taken. This implies the clause expression can be simply a method call that returns a boolean value. The syntax may be extended and as defined herein the equivalent behaviour can be supported by the specification of additional methods.

Syntax

```
conditional   : "IF" "(" clause ")" cbn_action
              | "IF" "(" clause ")" cbn_action "ELSE" "(" clause ")" cbn_action
              | "SELECT" [ "CASE" "(" clause ")" cbn_action ]+
              ;
clause :      target '.' method '(' parameters ')'
target :      STRING          {name of an object}
method :      STRING          {name of a method on the object.}
parameters:   parameter list  {list of names of other objects to be passed to
              the method}
```

Iteration Statement

At its basic level, an Iteration Clause could be viewed as being the same as a Condition Clause, i.e. if the clause equates to True then Iterate, otherwise stop.

OO programming allows the use of the *Iterator* object idea. If there is a collection of Objects one wishes to iterate across. This collection is given an order with a beginning, end, and a function for stepping through each Object from beginning to end. An Iterator is created, referring to the *beginning* Object, and is *stepped* through the collection until the *end* object is reached – the test therefore being whether the Iterator object refers to the end Object.

Syntax

```
iteration : "ITERATE" "(" clause ")" cbn_action ;
```

Guarded Statement

This construct is a way of synchronising threads. The current thread waits until the methods listed have been called. An object can only *guard* on its own methods. There are two choices as to how the Guard should work, illustrated by

the situation when the Guarded method occurs before the Guard is reached – what should happen?

1) The Guard was not blocking anything when the Guarded method occurred, and so should block when it is reached.
2) When the Guard is reached it should not block, as the Guarded method has occurred.

There is also the question of what to do if, for example, the guarded method occurs twice before the Guard is reached, and then the Guard is reached twice, or vice versa.

If we treat Guards in a similar fashion to semaphores, then we use a construct that is known about and widely recognised. A method occurring should be treated as a 'signal' to the semaphore when it ends, and a Guard on that method will be a 'wait' on the method's semaphore. This implements situation (2) above, and solves the following question.

Syntax

```
guard : "GUARD" methods ;

methods
  : String              { A method name. }
  | String "," methods  { A list of method names.}
  ;
```

Names

Globally, each object will have its own identity – it will be uniquely identifiable in some way – but may be referred to by a number of names depending on the context it is referred to in.

There are four ways to give an object its first name:-

1. From an external source setting up the global names.
2. As a contained attribute of a class.
3. As a parameter passed by value to a method.
4. As a result of a method call returning a new object, includes construction.

The name given in these cases, may not be all that is required in order to uniquely identify the object, but by including a selection of the names of the parent object, method, and platform (for distributed systems), a unique name can be made.

An object may be referred to by its global name, or by an alias to name, there are 3 ways in which an object may acquire an additional 'alias' name:-

1. As a reference attribute of a class.
2. As a parameter passed by reference to a method.
3. As a result of a method call returning a reference to an existing object.

These 'alias' names give an alternative handle on the object within a specific context, and may in some case become the only name for that object if the original name is lost, destroyed, or goes out of scope.

Interaction diagrams

Interaction diagrams give descriptions of sections of behaviour through a number of different objects, if the total of all these 'slices' of behaviour is taken, we can get a complete picture of the system behaviour, or rather, a complete picture of all of the behaviour that the designer is interested in, as this is all he/she has described in the interactions.

Looking at each object, in this total of interactions, the behaviour of that object under each stimulus that will be applied to it, can be seen, and hence the behaviour for each method for each object can be derived.

One purpose of the performance analyser is to provide performance modelling at an early stage in the design life cycle of a system and performance models of

'incomplete' designs, or more exactly, designs which do not have all the detail specified may be required. In particular this affects how the model chooses which path through the system to take. Conditional tests may not have enough detail in order to be evaluated, in which case the choice will have to depend on a probability.

The following describes how the interaction diagram constructs map to the behaviour constructs, using the UML collaboration diagram notation (though the mapping will work for sequence diagrams also). This mapping is valid in either direction, though to go from the behaviour actions to an interaction diagram, may need the introduction of additional methods on a class.

Method Call

This is a basic method call between two objects on a collaboration diagram, more complex behaviour is indicated by additional annotation. The call style is indicated by the style of the arrow head, and the result is given a name so that it can be used elsewhere in the diagram – as a parameter or as an object to which a message can be sent. There is no obvious way to show – within an interaction diagram – the cause or result of an exception.

gives:

```
SYNC result_name := method(parameters)
```

Sequential

gives:

```
SEQ {
  a.m1()
  b.m2()
  c.m3()
}
```

Sequential actions are indicated by sequence numbers. The hierarchical numbering system indicates two things: the hierarchy shows which messages are as a result of which other messages; and the numbers indicate the sequential order the messages are sent in at any particular hierarchy level.

Parallel

gives:

```
SEQ {
  a.m1()
  PAR {
    b.m2()
    c.m3()
  }
}
```

Parallel actions are indicated by the addition of characters. These are added after the sequence number at a particular level. Two messages having the same sequence number, but with an additional character indicating the difference, are assumed to be sent in parallel.

Iteration

gives:

```
SEQ {
  a.1()
  ITERATE(i = 1 to 10)
    b.m2()
  c.m3()
}
a::m1()
SEQ {
  ITERATE(j = 0 to 5)
```

```
    m4()
  }
  a::m4()
  SEQ {
    d.m5()
    e.m6()
  }
```

Iteration is indicated with a '*'. An iteration clause after the sequence number indicates that the message should be sent (or method called) repeatedly according to the clause. To iterate over several messages with the same clause, it can help to create a separate function in the calling object and the object should call this on itself.

Condition

gives:

```
SEQ {
  a.m1()
  IF (x<=7)
    b.m2()
  c.m3()
}
```

Conditions are very similar to iterations, the only difference being the lack of a '*'.

Guard

gives:

```
SEQ {
  a.m1()
  GUARD m3
    b.m2()
```

```
}
```

A guard on a message is indicating that the message will not be sent until the messages listed in the guard have been sent. It is preferable for a message number to be in a guard list, if that message will at some time be sent to the client object.

Probabilistic Condition

gives:

```
SEQ {
 IF (P(x,a))
  a.m1()
 IF (P(x,b))
  b.m2()
 IF (P(x,c))
  c.m3()
}
```

This is treated the same as a normal Conditional Statement, except for the semantics of the clause. Here, the clause is a pair of values – P(x,y) – interpreted as, the Probability that random variable 'x' has the value 'y'. Where x appears more than once in a single name space, it should be interpreted as the same variable tested against different values. The parameter 'y' may be a name which is given a value somewhere else, but must be a constant value. This can be seen as similar to a case or switch statement, giving:

```
SWITCH (P(x)) {
 CASE a: a.m1()
 CASE b: b.m1()
 CASE c: c.m1()
}
```

But such a switch statement can be built up from multiple if statements, as above, so it is unnecessary to have the additional construct.

Statecharts

If a class exhibits state like behaviour then its behaviour can be represented by a Statechart, however, Statecharts were not designed to represent class or object behaviour, and consequently classes with Statechart described behaviour have certain limitations. It is also necessary to define how the Statechart notation maps to class and object notation.

There have been a number of papers which address the issue of matching Statecharts up with Classes, two of which are by Harel (David Harel, Eran Gery; *Executable Object Modeling with Statecharts*; Proc. 18th Int. Conf. Soft. Eng., IEEE Press, March 1996) and Peach (Barbara Peach, Bernhard Rumpe; State Based Service Description; Formal Methods for Open Object-based Distributed Systems Volume 2; 1997). The Harel paper ties Statecharts to his own OO notation, and the Paech paper defines a new State machine like notation which ties in to the UML notation. Breu et al (see above) also talks about Statecharts, but again they have their own "similar state diagrams".

The performance analyser uses the Rose tool for input, so it is preferable that Statecharts be tied to the UML as these are the notations represented in the tool. The semantics are close to the method described in Harel, but recognises certain limitations which are backed up by observations made in the Paech paper.

Case studies seem to indicate that designers prefer the Interaction diagrams for specifying the behaviour of the system, and Statecharts are rarely used, consequently the performance analyser only allows for simple state-class semantics, however this could be expanded.

Statechart - Class Semantics

When an object is created, it starts in the starting state(s) indicated. Transition between states is via *events*, these events should map to method calls on the object. As the result of a transition an action can occur, this action can be a *send_event* which is the calling of a method on another object. Methods on such a class do not return a result, they just stimulate the object to change state.

Statechart to CBN Mapping

The Statechart describes the behaviour of the object. The behaviour seen when a method is called will depend upon which state the object is in. To model this in a CBN style, a state attribute is added to the class, and this is tested when a method is invoked, the result of that test indicates the consequent behaviour, some of which will be to set the state attribute to the new state. An event maps to a method call.

A transition maps to a conditional statement within a method. It has three parts: first, the test clause which tests the state attribute; second, the send_event caused by the transition if it is taken; and third assignment of the state attribute to the new state.

As an example here is the class description in the CBN syntax for the class described by the Statechart in Figure _____.

```
CLASS video_machine {
 String state

 play()
  SEQ {
   IF (state == "Stopped")
    SEQ {
     SYNC state.assign("Playing")
    }
   IF (state == "Play Fast")
    SEQ {
     SYNC state.assign("Playing")
    }
  }
 stop()
  SEQ {
```

```
    IF (state == "Play fast")
     SEQ {
      SYNC state.assign("Stopped")
     }
    IF (state == "Fast forward")
     SEQ {
      SYNC state.assign("Stopped")
     }
    IF (state == "Playing")
     SEQ {
      SYNC state.assign("Stopped")
     }
    IF (state == "Stopped")
     SEQ {
      ASYNC speaker.beep()
      SYNC state.assign("Stopped")
     }
   }
   ff()
   SEQ {
    IF (state == "Playing")
     SEQ {
      SYNC state.assign("Play fast")
     }
    IF (state == "Stopped")
     SEQ {
      SYNC state.assign("Fast forward")
     }
   }
}
```

Client-Cache-Server Case Study

The choices in this case can be considered independent, they are made upon the internal state of the "cache" – is the page request already stored in the cache, and is the date of the page stored in the cache fresh or not. Consequently the choice can be a probability contained within the cache, that determines the cache state when asked for a page – at this level of detail. A more detailed design of the system might see the probabilities distributed between cache and server, and some data values, like a date, and page_address introduced.

Notice that the use of synchronous method calls requires return statements to be included and numbered.

The Common Behaviour Notation for the CCS system is shown below, note that information from the Static Structure (or Class) Diagram is also required in order to generate this.

```
CLASS Server {
METHODS
 date date_request(page_address pa)
 SEQ {
  return ();
 }

 page request(page_address pa)
 SEQ {
  return ();
 }

ATTRIBUTES
};

CLASS Cache {
METHODS
 page request(page_address pa)
 SEQ {
  SELECT {
   CASE P(page_is,Uncached) :
    SEQ {
     SYNC m_server.request(page_address);
     return ();
    }
   CASE P(page_is,Fresh) :
    SEQ {
     SYNC m_server.date_request(page_address);
```

```
        return ();
       }
      CASE P(page_is,Stale) :
       SEQ {
        SYNC m_server.date_request(page_address);
        SYNC m_server.request(page_address);
        return ();
       }
      CASE P(page_is,Cached) :
       return ();
     }
    }

  ATTRIBUTES
   REF(Server) m_server;
  };

  CLASS Client {
  METHODS
   page request(page_address pa)
   SEQ {
    SYNC m_cache.request(page_address);
    return ();
   }

  ATTRIBUTES
   REF(Cache) m_cache;
  };
```

CMDS to SES

The conversion process from CMDS to SES will now be described.

The translation process from Composite Model Data Structure (CMDS) to SES performs a function that maps the relevant information in the CMDS into a performance model that is supported by SES Workbench. In order for the mapping function to progress from CMDS to SES, the performance analyser proceeds via an intermediary step which utilizes the SES Query Language. Use of this intermediary step supports automated generation of an SES model. The mapping function produces a query language file that is used by the query interpreter to generate an SES model giving the complete CMDS to SES translation process as illustrated in Figure ____.

The SES performance model produced by the translation has a structure which is referred to as a "Meta Model". The meta model is a framework within which a performance model of an object oriented (oo) system may be generated. The four layers of the input tools, namely Application, Workload, Execution Environment and System Scenario are explicitly modeled in the SES model produced by the translation process. The resultant SES model also makes use of predefined library components for Workload, Application and Execution Environment layer classes. These are SES representations of the library classes declared in the Rose source specifications for the Application and Execution Environment, and for the Workload objects declared in the Configurator System Scenario specification. In addition to the Workload, Application, Execution Environment and System Scenario, the translation process creates a fifth component in the performance model, a notional Object Request Broker (ORB) layer. This component provides management of inter-object communication for distributed Application layer objects.

The Meta Model

The meta model itself is an object oriented approach to performance modeling and describes the system to be modeled as a collection of objects that interact by means of method calls. The meta model represents the change in state and behaviour, over time, of the objects as a result of method calls made on them. All model elements are considered as objects including those from the Workload and Execution Environment layers, such as users, platforms, networks etc. For example, a PC might be seen as an object that has the method *process*, the behaviour of which is to delay for a period that is a function of the size of data requiring processing (a parameter passed to the method call) and the processing power of the PC object (an attribute of the object itself). Thus the meta model representation of a platform object, pc1, of class PC, processing an amount of data, data_size, for an Application layer object, a, of class AA, is illustrated by the object interaction diagram of Figure ____.

Similarly, the user of a web browser application object might be seen as an object of class User which makes the method call page_request(URL) on an object of class WebBrowser. These relationships are represented in the meta model with Workload/Application, Application/Execution Environment, and Execution Environment/Execution Environment objects interacting directly with one another. However, method calls between Application objects take place via the ORB layer which is seen to exist between the Application and Execution Environment layers. As mentioned previously, this layer is automatically added as part of the translation process in order to provide a mechanism to cope with the distributed nature of applications and application objects that are created and destroyed dynamically. The ORB layer consists a number of Object Adapter (OA) objects, one for each platform object, which are considered to "support" all of the Application layer objects on that platform. A method call between two Application layer objects is first made on the supporting OA for the calling object, which determines the location of the target object and associated platform. The method call *transmit* is then made by the OA on its associated platform, with the original method call name and its parameters the parameters of this new method call. Execution

Environment mechanisms then provide for the transportation of this information to the platform of the target Application layer object, and so the OA for the target object. The parameters of the transmit method call received by the OA of the target object are then used to make the required method call. Figure ____ illustrates how the interaction of two distributed Application layer objects a and b is realized by the meta model.

The transmission of method calls between platforms uses the network graph defined in the source system scenario and a routing table derived from it (Note: the network graph referred to here is the entire collection of individual execution environment objects rather than any individual network object (e.g. a LAN, WAN, etc.) which are considered to be nodes of the graph, and the arcs the connections between network objects specified in the system scenario). The routing table specifies a route from each platform in the system scenario to every other platform in the system scenario. Routes are automatically generated by finding the path containing the minimum number of nodes. For paths of the same "length", the route first found is used. Figure ____ shows a network graph and the routes generated for platform pca. Each route entry in the routing table consists of a list of platform and network object names. The first two names are the source and destination platform names whilst the remaining names are the names of the network objects (i.e. the graph nodes of Figure ____) that constitute the route. The route from pca to pcb would be:

The routing table is passed as a parameter to the SES model and is accessible by each platform object. A platform transmitting a method call uses unshared data to store the appropriate route declared in the routing table as a stack on the transaction. From the example of Figure ____ and given the network graph of Figure ____, the transaction transmitted by pca would appear as in Figure ____.

The route stack is used by the nodes in the network graph to direct the transmission transaction. At each node the top of the stack is popped and the new top used to select the arc taken to the next network node (see the section titled

"System Scenario Generation" below and Figure ___). Popping the top of the route stack of a transaction is the network functionality added to network objects as part of the translation process as described in the section titled "Execution Environment Generation" below.

The Translation

The meta model is realized using a multi-module structure, each module representing a different source model layer (and the additional ORB layer), and is driven by the system scenario module produced by the translation process. Thus the system scenario component is captured in an SES *Main Module*, whilst the Workload, Application, ORB and Execution Environment layers are represented in SES *Modules*. The library classes are defined in a separate SES catalog referred to as the CMDStoSES library which categorizes them as belonging to either Workloads, Applications, ORBs, Platforms or Networks, each of which is captured in an SES *Module*. The CMDStoSES library also contains an SES Module that defines global declarations for the performance model. Library class definitions and declarations are accessed by the meta model by declaring SES *Remote Modules* for each of the library modules. Thus the meta model catalog page appears as in Figure ___. (Note: Figure ___ also indicates two additional nodes in the catalog page. The first, an Archive icon, is used to include the C library functions that support the representation of dynamic objects, and routing mechanisms. The second, a Catalog Alias, is used to reference the CMDStoSES library.)

To summarize, the translation process generates the following meta model components which are contained within SES *Modules*:

- Workload.
- Application.
- ORBware.
- Execution Environment.

The translation process generates the system scenario meta model component which is used to drive the performance model, and is contained within and SES *Main Module*.

The translation process also generates references to library classes that have been defined in the CMDStoSES library catalog using SES *Modules*. The library classes are referenced using the *Remote Modules*:

- Workloads.
- Applications.
- ORBs
- Platforms.
- Networks.

A reference is also made to the library definitions for global declaration required by the model using the *Remote Module* LibraryDeclarations.

Dependencies between model components in terms of their SES modules are declared using SES *Dependence* arcs. The dependencies within the meta model are:

Module	Modules dependent upon
• SystemScenario.	Workload, Application, ORBware, Execution Environment, LibraryDeclarations.
• Workload.	Workloads, LibraryDeclarations.
• Application.	Applications, LibraryDeclarations.
• ORBware.	ORBs, LibraryDeclarations.
• Execution Environment.	Platforms, Networks, LibraryDeclarations.

System Scenario Generation

The SES system scenario is created using the corresponding system scenario description in the CMDS. The CMDS can be viewed as containing three major divisions of the information used by the translation process, as illustrated in Figure

___. (Note: each of the system scenarios has a corresponding route derived from its topology.)

Each of the System Scenarios within the CMDS is a list of the objects composing the given scenario. Each entry in the list contains values of object attributes and a link to the corresponding class description where appropriate (objects whose classes are defined in libraries do not contain these links, i.e. the links are only present for certain Application layer objects).

The objective of the translation process in creating the SES SystemScenario is similar to its purpose in the terms of the source tools in that it brings together the individual component definitions in the system (i.e. the objects) into a single configuration. Recall from the section titled "The Translation" that the translation process creates modules for each of the layers and within these are SES performance models for each of the objects of that layer, as specified in the Configurator system scenario. As such, these models exist as separate components and it is the function of the SES SystemScenario *main module* to link them together into single model. This is done by mirroring the system scenario configuration specified in the source tool using submodel *reference* nodes to refer to the appropriate components.

Within the SES SystemScenario *Main Module* the translation process generates a *submodel definition* node named to correspond to the CMDS system scenario. A second *submodel definition* node, "SystemScenarioInitialisation", is also created in the *Main Module* for use in initializing the model. Within the first of these nodes, a *submodel reference* node is created for each of the objects in the CMDS system scenario list belonging to the Workload and Execution Environment layers, and is named relevant to its CMDS source object (source object name may be prefixed by "RefTo_"). In addition a *submodel reference* node is created to represent each OA object, thus one is created for each *platform_instance* in the scenario list and named relevant to its associated platform ("RefTo_" + *platform name* + "OA"). As Application layer objects can be created and destroyed dynamically throughout the life cycle of a model, individual *submodel reference* nodes cannot be used to represent these objects as they cannot be created or destroyed during execution of

the model. Thus *submodel reference* nodes are created for each class of object that will be on a given platform, i.e. not only those objects explicitly indicated in the CMDS system scenario, but also those that are dynamically created during the execution of the model. These nodes are named using the convention: *name of class* + "_objects_on_" + *name of platform*.

Connections, using topology arcs, are made in both directions between Workload and Application layer nodes as indicated in the CMDS system scenario and between Execution Environment nodes. Similar direct connections are also made between Application layer nodes and their supporting platforms (again as indicated in the CMDS system scenario). Finally, the OA nodes are incorporated linking to their associated platform and the classes of objects which are supported by said platform (all of these connections are also made in both directions). Given the multiple arcs leading from nodes, the condition field of each arc is set. Recall that method calls between objects are represented by transactions, and the system scenario submodel represents the objects within the system. Thus the arc conditions use the name of the target object associated with a transaction.

As an example, consider the system scenario for a simple web page caching model (note: The Application layer description of this model can be found above) illustrated in Figure ____.

The translation process from CMDS to SES would produce a ***submodel definition*** node whose contents would correspond to this example as in Figure ____. According to SES convention, each reference node is labeled with its name followed by the name of the *submodel* node it references. Figure ____ also contains two additional nodes; the first is a source node used to populate the model with an initial transaction, whilst the second is a reference to the model initialization node mentioned above.

The submodel SystemScenarioInitialisation initializes the model by stimulating the creation of all those Application layer objects that initially populate the system as indicated by the CMDS system scenario. Thus the SystemScenarioInitialisation submodel uses the single transaction generated by the *Source* node "WLSource"

(see Figure ____ (3.4)) to create additional transactions for each object to be created. Each new transaction represents a *Constructor* method call to create the new object and has as parameters for the attribute values of the object. These attribute values are supplied by a parameter file created as part of the translation process from the attribute values supplied in the CMDS system scenario. The constructor transactions are then routed to the appropriate OA, via the additional topology arcs from the SystemScenarioInitialisation submodel reference to each OA reference, and then on to the object class submodel reference.

The SystemScenarioInitialisation submodel also uses the single "WLSource" transaction to stimulate the Workload layer objects. Similarly, additional transactions are spawned for each Workload layer object referenced in the SystemScenario submodel and proceed directly to them from the SystemScenarioInitialisation submodel reference.

Workload Generation

The Workload module contains SES models of all the Workload layer objects indicated in the CMDS system scenario list. Since all workload classes are defined to be library components, the SES models for them are contained within the CMDStoSES library module "Workloads" and referenced in the performance model catalog page by a *remote module* (see the section titled "The Translation"). Each workload class definition in the "Workloads" library is contained within a *submodel type* node, and individual instances of these are declared for use in the performance model generated by the translation process. Thus the Workload module generated by the translation process simply contents *submodel instance* nodes for each of the workload objects in the CMDS system scenario list. The name of each node is the same as its CMDS name, and its type is set to the *submodel type* workload model of the CMDStoSES library that corresponds to the library component class declared in the CMDS.

The caching example of Figure ____ is expanded further in Figure ____ to illustrate the Workload module and contents that are created by the translation process and the relationship to the library module and the CMDS. From Figure ____ it can be

seen that the translation process generates a module "Workload" whose contents consist of a single *submodel instance* node "u1" which is an instance of the *submodel type*, "User". The submodel type "User" is declared in the CMDStoSES library module "Workloads" which is referenced using an SES *remote module* (note that the workload library module might also contain other workload types as indicated by the additional submodel type nodes "ClosedWL" and "OpenWL"). In relation to the CMDS it can be seen that the instance node "u1" of node type "User", directly maps to the object "u1" whose class (or type) is User.

ORB Generation

Generation of the ORB layer and its constituent OA objects is a simplified version of the workload generation process of the section titled "Workload Generation" above. The simplification is a result of inserting the ORB layer as opposed to allowing users to input its specification. As a result all OA objects that are created are of a single type which is declared in the CMDStoSES library. The only relationship between the contents of the ORB layer and the CMDS is that an OA object is created for each platform_instance in the CMDS system scenario list. The ORB layer generated for the caching example is shown in Figure .

Default Object Adapter

The ORB layer of the meta model is comprised solely of Object Adapters (OA), of which there is one for each Execution Environment platform. OAs are the mechanism that enable Application layer objects to communicate with one another without any knowledge of how the system is distributed. OAs use and manage two tables of information that register which Application layer objects are supported by which OA, and which OAs are associated with which Execution Environment platform. Thus for the simple web page caching model, again illustrated in **Error! Reference source not found.**, the two ORB layer tables are given by tables 1.1 and 1.2.

Application Object	ORBware OA
cl1	pc1OA
ch1	lcOA
svr	rsOA

Table 1.1: Objects and ORBs

ORBware OA	Execution Environment Platform
pc1OA	pc1
lcOA	lc
rsOA	rs

Table 1.2: ORBS and Platforms

The functionality of the default OA is illustrated in Figure ____. A transaction entering the OA represents a method call made by an Application layer object. The OA first determines whether the calling object is local (i.e. on the same platform and thus has the same OA) or remote. This is done by comparing its name with that of the OA supporting the calling object (determined from the information stored in Table 1.1). The appropriate arc, "Caller Local" or "Caller Remote", can then be taken. Note the third arc is used during the initialization of the model to create the Application layer objects that initially populate the model since no objects exist to make these constructor calls.

For method calls from local objects (i.e. taking the "Caller Local" arc), the methods are categorized as either, *Constructor*, *Destructor*, or *other*. Constructor method calls proceed to the *UpdateORBTable* node where tables 1.1. and 1.2 are updated and then to the *ToLocalFromLocal* which forwards the transaction onto the Application layer to create the object itself (note this route is also taken during the initialization process as mentioned in the preceding paragraph). Destructor calls proceed to the *UpdateTables* node where the entries for the appropriate object is removed from tables 1.1 and 1.2. For the remaining method calls the supplier (or target object) is determined to be either local or remote and the appropriate arc taken (i.e. "Supplier Local" or "Supplier Remote", respectively). For local suppliers, the transaction simply leaves the OA. For remote suppliers the method call is packaged in the *ToRemoteFromLocal* node, up as in Figure ____ of the translation process document before exiting.

For method calls from remote objects (i.e. taking the "Caller Remote" arc), the methods can be neither *Constructor* or *Destructor* thus it is only necessary to check that the supplier (or target object) is local. For local suppliers (i.e. the "Supplier Local" arc), the method call is unpackaged in the *ToLocalFromRemote* node, as in Figure 2.2 of the translation process document before exiting. If the supplier is not local (i.e. the "Wrong ORB" arc is taken) then an error has occurred, and a notification is made.

Execution Environment Generation

As with the Workload and ORB layers (see sections titled "Workload Generation" and "ORB Generation" above) all objects from the Execution Environment layer are declared to be instances of library classes and as such have definitions in the SEStoCMDS library. However, when incorporating instances into the performance model generated by the translation process these class descriptions are augmented by additional SES constructs in order to interface with the meta model structure.

Execution Environment *Platform_Instances* within the source model are primarily to provide processing support for Application layer objects and as such the SEStoCMDS library definitions of platform types are simple data processing models. However, within the meta model structure, a platform plays additional roles and thus the translation process generates the required functionality. The additional roles performed by a platform in the meta model are concerned with network interfacing and statistics collection.

A platform supports two functions for network interfacing; the transmission and receipt of messages. The first of these is as a result of a transmit method call on the platform object made by its associated OA object (see the Section titled "The Meta Model" and Figure ___). A route from the platform to the destination platform is determined and associated with the transaction (recall that the transaction represents the method call). The second function, again as the result of a transmit method, occurs when the transmission reaches the destination platform. The destination platform removes any remaining routing information, converts

transmit method call parameters into the original method call (see the section titled "The Meta Model" and Figure ____) and forwards it to its associated OA object.

Statistics collection functionality is generated in order to interface with the statistics collection that is performed for the rest of the model and to enable library components to be created independently of any knowledge of how statistics are collected for the performance model.

An illustration of the additional network and statistics interfacing functionality added to a platform object is illustrated in Figure ____.

Network objects require less augmentation in order to interface with the meta model as they only perform the single role of modeling transmission across a network. Thus only the statistics collection and limited routing functionality is added to network objects created by the translation process. The routing function is simply to indicate which is the next network object in the route specified between source and destination platform objects. These additional functions are illustrated in Figure ____.

The translation process generates a *submodel definition* node in the Execution Environment module for each Execution Environment layer object specified in the CMDS system scenario and named accordingly. A *submodel instance* node is also created for each Execution Environment layer object specified in the CMDS system scenario and named accordingly with the postfix "Processor". The type of the instance node is set to the platform or network model *submodel type* of the CMDStoSES library that corresponds to the library component class declared in the CMDS. A reference to the instance node is used within the definition node in order to incorporate the library definition of the data processing or network transmission (depending on whether a platform or network object is being modeled) model component. Thus the nodes within Figure ____ and Figure ____ indicated to represent the library definitions are *submodel reference* nodes to the submodel instances of the library submodel types. Figure ____ shows the Execution Environment layer generated for the caching example, the platform object pc1 has been expanded to show the relationship between the submodel definition and

instance nodes necessary to model a single platform, and their relationship with the library model submodel type.

Application Generation

For the Workload, ORB and Execution Environment layers, the CMDS to SES translation process generates a performance model object for each object in the system scenario list (see the sections titled "Workload Generation", "ORB Generation" and "Execution Environment Generation" above) since dynamic creation of objects from these layers is not permitted. However, the meta model of the Application layer does supports the dynamic creation and destruction of objects. Since it is not possible to dynamically create and destroy SES graphical components, Application layer objects are created and represented by the translation process differently to that of objects from the other three layers.

Application layer objects are represented using two components; an SES component which describes the logical behaviour of objects of that class which is derived from the Common Behaviour Notation (CBN) (above), and a dynamic data structure component which describes the identity and state of each object. Only one SES component is created for each class and so a single behavioural description is used by all objects of that class. An individual data structure is created for each object that is created and the structures for all objects of the same class are stored collectively in a single dynamic table, or Class Object Table (COT). Thus, for each Application layer class the translation process generates an SES model of the behaviour of the class from the CBN and a COT in which identity and state information of each object of that class may be stored. The table is a simple linked list implemented in the C programming language enabling new objects to be added and deleted throughout the lifetime of the simulation. Similarly, the structures used to store the identity and state of individual objects are also linked lists in order to provide generic support for objects of different classes which may have differing numbers of attributes. The Application layer references made in the SystemScenario main module (see the Section titled "System Scenario Generation" and Figure ____) refer to the SES class behaviour models detailed above.

Application Layer Class Behaviour

Concurrency Control

The class behaviour generated by the translation process is a graphical representation of the CBN description in the CMDS. However, the CBN description does not extend to concurrency control, and in order to manage concurrency control dynamic structures similar to those used to represent the state of an object are used in conjunction with the afore mentioned SES construct to correctly manage the dynamic nature of Application layer objects. These structures are similar to the COT but consist of concurrency control queues for each object of a given class and are referred to as Class Queue Tables (CQT).

The source specification of an Application layer class indicates the number of concurrent threads that an object of that type may support (i.e. the number of method calls the object may process concurrently). Any threads (or method calls) in excess of this are queued by the object until such time that one or more threads is freed (i.e. the number of active threads in the object is less than its maximum capacity). Recall that method calls are modeled by transactions, thus any transaction entering the class behaviour submodel as a result of an external method call (i.e. a method call made by another object) is queued if the object is currently busy. Since the class behaviour is a static graph but Application layer objects are dynamic, the transactions cannot be queued using an SES node for each object. Instead, the identity of queued transactions are recorded in the appropriate queue of the CQT, whilst the transaction itself is queued in a single *blocking node* at the beginning of the class behaviour graph. The object queues of the CQT preserve the order in which method calls are to be serviced by individual objects, whilst the blocking node provides the simulation mechanism to queue the transactions themselves (the blocking condition is always false). When a queued method call can be serviced, the table queue is used to identify the appropriate transaction and an *interrupt node* frees it from the *blocking node*.

Figure ____ shows how the concurrency of three objects *a1*, *a2* and *a3* of class A is handled. Method calls have been made on object *a1* using transactions T8, T4

and T2, and arrive at the object in that order. The method calls and their order of arrival can also be seen for objects *a2* and *a3*. The order of arrival of method calls at the three objects collectively can be seen by the single queue of the blocking node in the behaviour model of class A. From the queues for each of the three objects, the correct transaction can be identified and freed from the collective blocking node queue when a given object becomes free.

Following the concurrency control segment the class behaviour graph branches into segments representing the behaviour of each of the methods of the class. The ends of these branches then rejoin at an *interrupt* node to form a single flow of control before exiting the class (in the case of a method call made synchronous) or terminating in a *sink* node (in the case a method call made asynchronously). A transaction reaching the end of a method call will either leave the object (a synchronous call) or terminate (an asynchronous call) making a thread available. The interrupt node at the end of the class methods determines which transaction to assign this thread to by use of the object's name and the CQT (see Figure ___) and interrupts it from the blocking node at the beginning of the class behaviour graph. This is illustrated in Figure ___ which shows the behaviour for a class that has two methods f() and g(). (Note: for implementation reasons, there are a separate sink and interrupt nodes for methods called asynchronous and a single interrupt and return node for methods called synchronous.)

Parameter Passing

Parameter passing is supported by using a dynamic list that is linked to a transaction by an ushared variable. Parameters always refer to objects and may be passed by reference or value, with the size of the parameter set appropriately to represent value or reference. Passing a parameter by value requires a copy of the named object to be made and passed as the parameter. This is done by the SES construct representing the calling method (see the sections titled "Concurrency Control" and the section titled "Method Behaviour") which copies the object in the COT (see the section titled "Application Generation") which assigns the new object a unique name. This new name is then passed as the parameter value.

The platform on which an Application layer object resides is stored as an attribute of the object, and for objects created using constructor calls is determined at the time of creation. However, for objects created as the result of passing a parameter by value, this is resolved when the parameter reaches the target object. Similarly, when the called method is completed, any objects passed by value are considered to be out of scope and thus must be deleted. These two activities are conducted at the beginning and end, respectively, of the method behaviour as illustrated in Figure ____. The localise parameters node performs the task of setting the platform attribute of those parameters passed by value to the current platform. (Note: For those methods that do not have parameters passed by value, there is no need for this functionality, and for implementation reasons, the node is replaced by a branch node.) A delete parameters node appears at the end of each method and removes those parameters passed by value from their respective object tables. This node also performs the task of returning synchronous method calls when no return method is explicitly declared (see the section titled "Method Call (Return)"). Deleting parameters passed by value is also performed by return method call constructs (see the section titled "Method Call (Return)") since the return method call path bypasses the default delete parameters node of Figure ____.

Method Behaviour

The behaviour of a class method is described in the CMDS using a set of CBN constructs. These constructs have parallels in the SES class behaviour model generated by the translation process. The translation process generates the behaviour of a method by recursively iterating over the CBN description in the CMDS and generating the corresponding SES description. Each of these descriptions are linked by topology arcs as they are created. The CBN constructs and their SES equivalents are described below.

Method Call (Synchronous)

SYNC result_name : = target.method(parameters)

The translation process uses *fork* and *join* nodes to model CBN synchronous method calls. A single child transaction is spawned and represents the method call whilst the parent proceeds directly to the *join* node and waits for the return of the child transaction (the return of the synchronous call). The child transaction has unshared variables that contain target and parameter information. The child transaction has two possible routes when leaving the *fork* node; the first leads to the exit node of the class (see also Figure ___) and is taken when the target is not *THIS*, the second is taken when the target is *THIS*, and leads to the start of the method of the class beahviour (see also Figure ___). Since synchronous method calls suspend the behaviour of an object (i.e. the parent goes directly to the *join* node and awaits the return of the child transaction), calls made on *THIS* bypass the concurrency control for the object as it is considered that the thread of control is passed directly to the new method call (i.e. the call on *THIS*).

Returning child transactions are routed directly from the start of the class behaviour to the *join* node bypassing the concurrency control as the thread is deemed to return to the parent transaction when joined by its child. Each of the nodes in a class behaviour are uniquely named. The name of the join node is stored in an unshared variable of the child transaction in order to select the correct arc when returning (there are return arcs for each synchronous method call, named according to the node returned to).

Synchronous calls return objects by value or reference as their result. The name of the object returned is assigned to *result_name* which may be an attribute of the calling object (and as such stored in the COT - see the section titled "Application Generation") or a parameter of the method within which the method call was made (in which case the return value is assigned in the parameter list of the parent transaction).

Method Call (Asynchronous)

ASYNC target.method(parameters)

Method calls made asynchronously generate a new thread independent of the originating thread. As such, the translation process uses a *split* node to represent CBN asynchronous method calls which spawns a single sibling transaction for the new thread. The sibling method call transaction has calling and parameter information stored in unshared variables and as with synchronous calls can take one of two possible routes from the *split* node. One route again leads to the class exit node and is taken if the target object is not *THIS*. Given that an asynchronous call creates a new thread, the new sibling transaction cannot bypass the concurrency control of an object as in a method call made synchronously. Hence, the second path leads to the start of the class behaviour, *before* the concurrency control.

Note that transactions spawned by both synchronous and asynchronous calls inherit the call path of the parent transaction in order to enable the new transaction to leave the current class behaviour submodel.

Method Call (Return)

The return method call is translated into a *user* node which simply switches the caller and target unshared variables of the calling transaction, and sets the method call name unshared variable to return. The return node also sets the return parameter to that value specified by the CBN statement.

The CBN description of a method may not explicitly specify a return method call but may still be called synchronously. Thus a default return is declared for each method call, and is added to the parameter management node at the end of each method (see the section titled "Parameter Passing" and Figure ____). A default return does not return any value, it simply returns the calling transaction to the source object.

Conditional

IF (boolean condition)
CBN statement

The translation process utilizes a synchronous method call (see the section titled "Method Call (Synchronous)") to implement the conditional component of this CBN construct. A synchronous method call which returns a boolean object by value is made and the returned value tested for true or false, the outcome of which is then used to route the transaction accordingly. Any arbitrary CBN statement may be declared for execution if the condition is true. On completion of this statement, or if the condition evaluates to false, the transaction proceeds to the remaining method behaviour.

Probabilistic Condition

```
SEQ {
        IF (P(x, a))
            CBN Statement
        IF (P(x, b))
            CBN Statement
        IF (P(x, c))
            CBN Statement
}
```

The conditional clause for a CBN *Probabilistic Condition* is a simple probability value, in this case one of values a, b or c. The translation process simply generates the SES representations of the CBN statements to be executed which are connected in parallel from the condition start *branch* node. The probability selection field of the arc specification form is set to the probability value in the corresponding CBN condition statement. Exit arcs from the end of the executed statements are made in parallel to the condition end *branch* node.

Sequential

```
SEQ {
        CBN statement
        CBN statement
        CBN statement
}
```

The CBN sequential construct is simply translated into the components of the statement linked in series by a topology arc as in Figure ____.

Guard

GUARD method
CBN Statement

The guard statement is implemented using a form of semaphore mechanism. Each method of an object has an associated counter which is incremented each time the method is called. Guarding on the method will block if the counter is less than or equal to zero. If the counter is greater than zero, the guard will not block and the counter will be decremented by one. In addition to the counter, each object method has a queue which records the identity of all those transactions blocked as a result of guarding on it. A blocked transaction is freed when the method be guarding on has been called and completed its execution. When a transaction is blocked as a result of guarding on a method, it releases its thread of control, and interrupts the appropriate transaction held in the concurrency control node.

Testing the counter and interrupting the concurrency control node is performed by a *super* node at the start of the guard behaviour. If the transaction is blocked by the guard, it proceeds to the guard *block* node, or the end of the guard behaviour otherwise. The guard behaviour is shown in Figure ____.

Parallel

Similarly, the parallel CBN construct may be supported by the translation process.

Iteration

Similarly, the iteration CBN construct may be supported by the translation process.

It is not of course necessary to take every aspect of the embodiment described above to build a performance analyser according to an embodiment of the present invention. In particular, it would be possible to use state transition diagrams in place of the interaction diagrams, although these would have to translated in a different way. It is not essential to use a library built in the SES environment. It would be possible to build in definitions using the source tools which are subsequently translated. It would be possible to substitute a language other than UML, such as Booch notation, although it should preferably be object based since this lends itself particularly well to the construction of the CMDS data. looping round *Next_Action* until all calls spawned.

(4) A synchronous action will return a value which may necessitate transmission over the network so must therefore follow *Return_Path_or_Call_Self* after returning from the called object submodel for this to occur.

Workload Module

In the source tool, and subsequently the CMDS, workload may be defined by a series of usage models. Each usage model relates to a particular application object which can accept external stimuli. These stimuli come in the form of events which change the state of the object according to its state chart.

Each type of event has a distribution determining the inter-arrival times between events of that type. Usage models may cover more than one type of event although each even type is independent and executes a single step at the workload level.

Inside SES/*workbench*, a submodel is created for each externally stimulated object in the workload module. The submodel has a source node for each type of event which has an inter-arrival time given by the corresponding distribution. The resulting source streams for each type of event all join to a node which references the object in question. When a transaction returns from the application, it is terminated.

Figure 41 shows an example workload submodel for a client_object application submodel. The source nodes, *request_source_0* and *refresh_1*, are so

Annex to CMDS to SES Translation Process

Statistics Collection

The statistics collected can be categorised as method call duration and platform and network utilisation/response. Due to the dynamic nature of the objects, collection and identification of statistics for individual objects within the meta model are preferably determined, for the duration of method calls, by time stamping each method call transaction when it is created, and comparing this to the time when it returns to its *join* node, in the case of a synchronous call (see Section 3.5.1.3.1), or when it reaches a sink node, in the case of an asynchronous call (see Section 3.5.1.3.2 and Figure 3.11).

Platform and network utilisation/response statistics are collected within the platform and network definitions themselves, as illustrated in figures 3.7 and 3.8. Each transaction is time stamped in a *user* node before it enters the library definition node. This time can then be compared in a second *user* node with the current time after it has left the library definition node in order to determine response and utilisation statistics.

Network Transmission

Recall from Figure 2.4, that each transaction that is transmitted between platforms has routing information associated with it in the form of a stack each element of which is a node on the route. However, it is preferable that the stack of Figure 2.4 also includes the name of the destination platform which is at the base of the stack, thus Figure 2.4 should be amended as shown in 1.1.

The network transmission functionality in the user node of Figure 3.7 generates the appropriate route stack for a transaction to be transmitted between two platforms. To represent transmission of a transaction, the top of the stack is used to determine the next node in the route. At each node, the top of the stack is removed. In this way the transaction traverses a route from source to destination platform. The network transmission functionality of Figure 3.8 performs the pop operation on the transaction route stack. A similar operation is also performed by the network reception functionality of a platform as illustrated in Figure 3.7. This

functionality may be generate within the C programming language and within SES associated with a form.

CLAIMS

1. A method of generating performance models of information technology or communications systems comprising the steps of:

1) inputting to modelling apparatus at least a first data set of operational information of said system;

2) inputting to said modelling apparatus at least a second data set of topological information of said system's architecture;

3) transforming said first and second data sets into at least one performance model of said system.

2. A method as claimed in claim 1 wherein said at least a first data set comprises a software application and wherein said method further comprises the step of:

(4) extracting said operational information from said software application.

1. A method as claimed in claim 1 or claim 2 wherein said at least a second data set comprises a definition of said system's architecture and wherein said method further comprises the step of:

(5) extracting said topological information from said definition.

4. A method as claimed in any preceding claim wherein said method further comprises the steps of:

(6) extracting at least a first set of interface data from said first and second data sets;

(7) linking said first set of interface data with a predefined interface whereby operatively associating said first and second data sets.

5. A method as claimed in claim 4 wherein step 6) further comprises the steps of:

(8) identifying inter-related sub-sets of data within said operational information;

(9) deriving at least part of said interface data from said identified sub-sets.

6. A method as claimed in claim 5 wherein step 7) further comprises the step:

10) associating at least one node of said topological information with at least one of said sub-sets of data.

7. A method as claimed in claim 6 further comprising the step of:

11) generating a unique identifier for each association between a node and one of said sub-sets of data.

8. A method as claimed in claim 5, 6 or 7 wherein step 6 further comprises the steps of:

12) identifying routes between nodes in said topological information;

13) deriving at least a part of said interface data from said routes.

9. A method as claimed in claim 8 wherein step 12) further comprises the step of a breadth first search of said topological data.

10. A method as claimed in any preceding claim wherein step 3) further comprises the step of:

14) associating said first and second data sets with at least one predefined data set.

11. A method as claimed in claim 10 wherein said first and second data sets comprise reference data and wherein step 14) further comprises the step of:

15) linking said reference data with said predefined data set.

12. Apparatus for generating performance models of an information technology or communications system comprising;

input means for receiving as input at least a first data set of operational information of a software application;

input means for receiving as input at least a second data set of topological information of said system's architecture;

transformation means for transforming said first and second data sets into at least one performance model of said system.

13. Apparatus as claimed in claim 12 further comprises first extraction means for extracting said operational information from a software application that in use formed input to said input means.

14. Apparatus as claimed in claim 13 or 15 wherein said first extraction means is further adapted to extract said topological information from architecture design information that in use formed input to said input means.

15. Apparatus as claimed in any one of claims 12, 13 or 14 wherein said apparatus further comprises interface means for interfacing said first and second data sets, second extraction means for extracting at least a first set of interface data from said first and second data sets, and association means for associating said interface data with said interface means whereby operatively associating said first data and said second data sets.

16. Apparatus as claimed in claim 15 further comprising identification means for identifying inter-related sub-sets of data within said operational information wherein said second extraction means is adapted to extract said interface data at least in part from output of said identification means.

17. Apparatus as claimed in claim 16 wherein said association means is further adapted to associate at least one node of said topological information with at least one of said sub-sets of data.

18. Apparatus as claimed in claim 17 further comprising naming means adapted to generate a unique name for each said association between a node and sub-set of data.

19. Apparatus as claimed in any one of claims 15, 16, 16, or 18 further comprising route identification means adapted to identify route data for at least one route between at least two nodes of said topological data.

20. Apparatus as claimed in claim 19 wherein said route identification means is adapted to identify said route data with a breadth first search of said topological information.

21. Apparatus as claimed in any one of claims 12 to 20 further comprising at least one predefined data set and means adapted to associate at least one of said first or second data sets with said predefined data set.

22. Apparatus as claimed in claim 21 wherein in use at least one of said first or second data sets comprise reference data and wherein in use said apparatus is adapted to link said reference data with said predefined data set.

23. A method of predicting performance characteristics of information technology or communications systems comprising the steps of:

(16) executing a performance model according to any one of claims 1 to 11.

24. A method as claimed in claim 23 step 1) further comprises the step of:

(17) associating said performance model with at least one predefined performance model component.

25. A method as claimed in claim 23 wherein said performance model contains at least one set of reference data and wherein step 2) further comprises the step of:

(18) linking said at least one set of reference data with said at least one predetermined performance model component.

26. Apparatus for predicting performance characteristics of information technology or communications systems comprising apparatus as claimed in any one of claims 12 to 20 and adapted to execute said performance model.

27. Apparatus as claimed in claim 26 further comprising at least one predefined performance model component and adapted to in use link said at least one predefined performance model component with said performance model.

28. Apparatus as claimed in claim 27 wherein in use said performance model is generated with reference data and wherein said apparatus is further adapted to link said reference data to said predefined performance model component.

29. System performance analysis apparatus for analysing performance in an information technology system, the apparatus comprising:

i) a data store for storing at least a first model of a first aspect of the system and at least a second model of a second aspect of the system;

ii) means for extracting or generating, and storing, mapping data for mapping related aspects of the first and second models;

iii) transformation means for transforming said first and second models, using said mapping data, into a data structure for use in generating a performance model of the system.

30. Apparatus according to Claim 29 which further comprises input means for inputting to the data store performance analysis results generated by performance testing means for performance testing the system, using said performance model.

31. Apparatus according to any one of claims 29 or 30 wherein said data store is adapted to store a third model of a third aspect of the system, said means for extracting or generating, and storing, mapping data is adapted for mapping related aspects of the first, second and third models, and said transformation means is adapted for transforming said first, second and third models, using said mapping data, into a data structure for use in generating a performance model of the system.

32. Apparatus according to Claim 30, which further comprises means for generating, and storing in the data store, link data for linking said performance analysis results to aspects of any one or more of said models of aspects of the system.

33. Apparatus according to any one of Claims 29 to 32, wherein the data store is adapted to store each of the models of aspects of the system in a common data structure

34. Apparatus according to Claim 33, the apparatus further comprising input means for receiving at least one model of an aspect of the system and for converting it to said common data structure for storage in the data store.

35. Apparatus according to any one of Claims 29 to 34, wherein said models of aspects of the system comprise an applications model, an execution environment model and a workload model.

36. Apparatus according to either one of Claims 34 or 35 wherein the common data structure comprises software objects and software object classes, each software object comprising a set of data in combination with one or more related operations, and each class comprising an object description which is potentially common to a set of non-identical objects.

37. Apparatus according to any one of Claims 29 to 36, wherein the transformation means comprises means for generating at least one component of the performance model by selecting a template from a library of templates and populating the template with data stored in the data store.

38. Apparatus according to Claim 37 which further comprises means for generating and loading templates to said library.

39. Apparatus according to Claim 36 wherein the data store stores, in respect of at least one model of an aspect of the system:

i) a class diagram to show the existence of object classes and their relationships in a logical view of the system;

ii) an object diagram to show the existence of software objects, their relationships and interactions;

iii) an interaction diagram to trace, in the system, the execution of a selected use of the system in time, for instance as a sequence of messages between objects of the system; and

iv) a state transition diagram to specify the detailed behaviour of any one or more of a class, a cluster of classes, and the system.

40. A method of analysing performance in an information technology system, which method comprises the steps of:

i) storing in a data store at least a first model of a first aspect of the system and at least a second model of a second aspect of the system;

ii) extracting or generating, and storing in the data store, linkage information for linking the first and second models;

iii) transforming said first and second models, using said linkage information, into a performance model of the system; and

iv) performance testing the system, using said performance model.

41. A method according to Claim 40, wherein at least one of the models comprises an interaction diagram to trace, in the system, the execution of a selected use of the system in time, for instance as a sequence of messages between objects.

42. A method according to Claim 41 which further comprises the modification of said interaction diagram and repetition of steps i) to iv).

43. A method according to any one of Claims 40, 41 or 42 wherein said step iii) comprises the steps of :

a) linking the first and second models together;

b) generating a state-based view of the system from scenario-based information;

c) generating a set of tables which represent states, events and actions in the system;

d) generating a performance model structured to correspond to the first and second model domains; and

e) building and executing the performance model.

44. A method according to any one of Claims 40 to 43 wherein the performance model is embodied in Information Processing Graph notation.

45. A method according to any one of Claims 40 to 44 wherein step i) further comprises storing in the data store a third model of a third aspect of the system.

Fig.1.

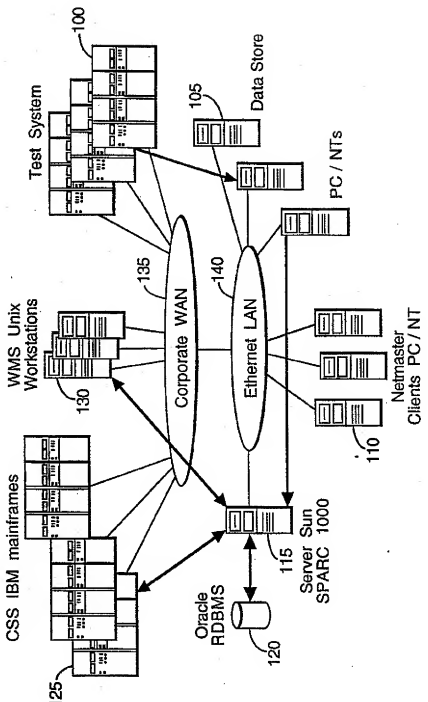

Test System
100
Data Store
105
PC / NTs
WMS Unix Workstations
130
135
Corporate WAN
140
Ethernet LAN
Netmaster Clients PC / NT
110
CSS IBM mainframes
125
Oracle RDBMS
120
115
Server Sun SPARC 1000

Fig.2.

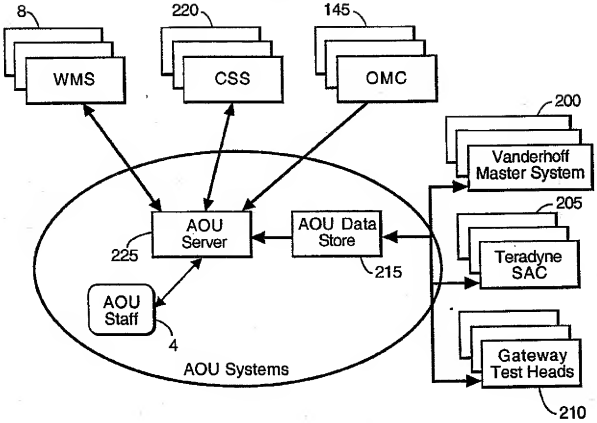
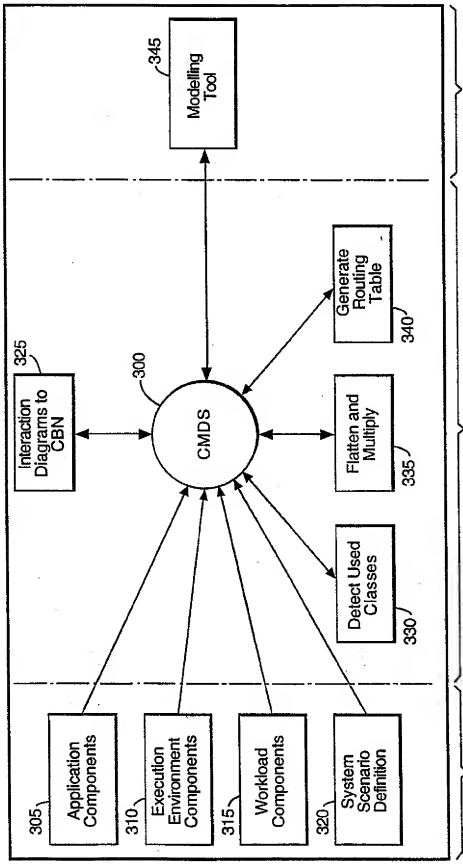

8
220
145
WMS
CSS
OMC
200
Vanderhoff Master System
205
AOU Server
AOU Data Store
225
215
Teradyne SAC
AOU Staff
4
AOU Systems
Gateway Test Heads
210

Fig.3.


Performance Analyser Architecture
Section 1
Section 2
Section 3
305
Application Components
310
Execution Environment Components
315
Workload Components
320
System Scenario Definition
325
Interaction Diagrams to CBN
300
CMDS
330
Detect Used Classes
335
Flatten and Multiply
340
Generate Routing Table
345
Modelling Tool

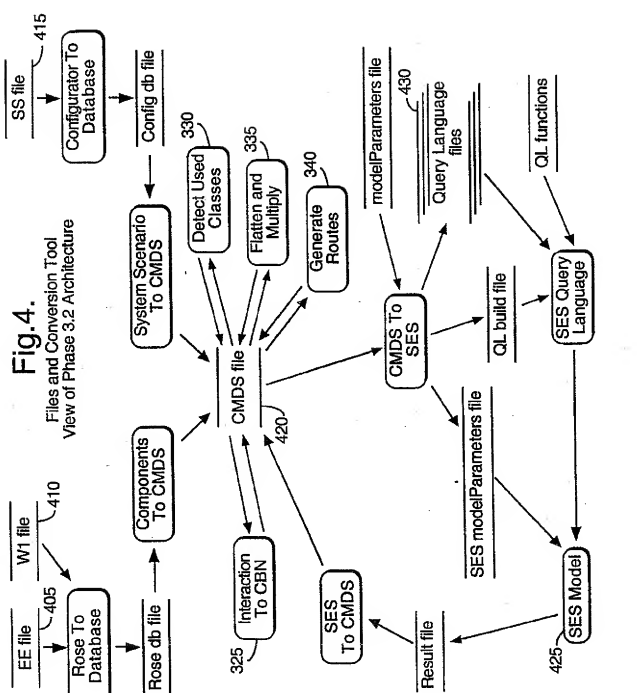


Fig.4.

Files and Conversion Tool
View of Phase 3.2 Architecture

Fig.5.

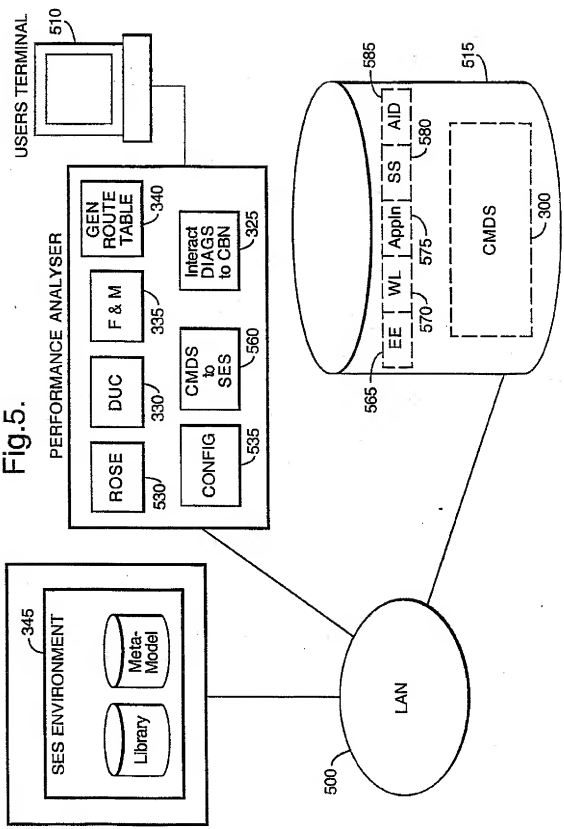

USERS TERMINAL
510
PERFORMANCE ANALYSER
ROSE
530
DUC
330
CMDS to SES
560
F & M
335
GEN ROUTE TABLE
340
CONFIG
535
Interact DIAGS to CBN
325
SES ENVIRONMENT
345
Library
Meta-Model
LAN
500
565
EE
570
WL
575
Appln
580
SS
AID
585
CMDS
300
515

Fig.6.

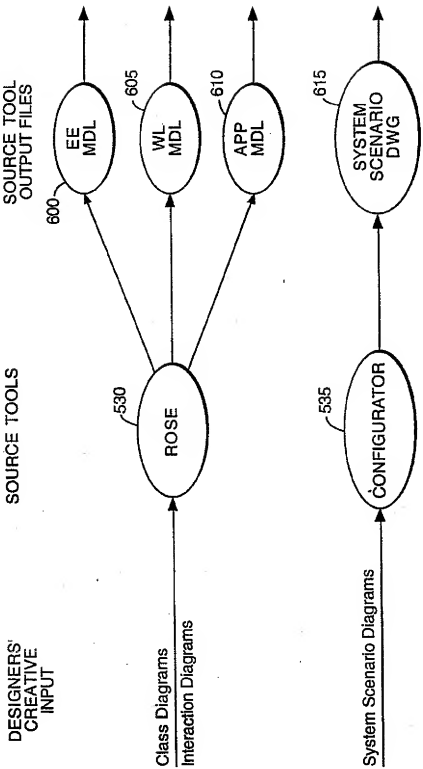

DESIGNERS' CREATIVE INPUT
SOURCE TOOLS
SOURCE TOOL OUTPUT FILES
Class Diagrams
Interaction Diagrams
ROSE
530
EE MDL
600
WL MDL
605
APP MDL
610
System Scenario Diagrams
CONFIGURATOR
535
SYSTEM SCENARIO DWG
615

Fig.7.

PERMABASE "DATABASE FORMAT"
CMDS1
CMDS2
CMDS3
CMDS4
EE Classes
WL Classes
APP Classes
APP Interaction diagrams
SYSTEM SCENARIO DWG
565
570
575
580
585
CLASSES
CBN
LINKED OBJECT STRUCTURES
ROUTING TABLE
300
1.
2.
3.
4.
1. CBN Conversion
2. Detect Used Classes
3. Flatten and Multiply
4. Generate Routing Table

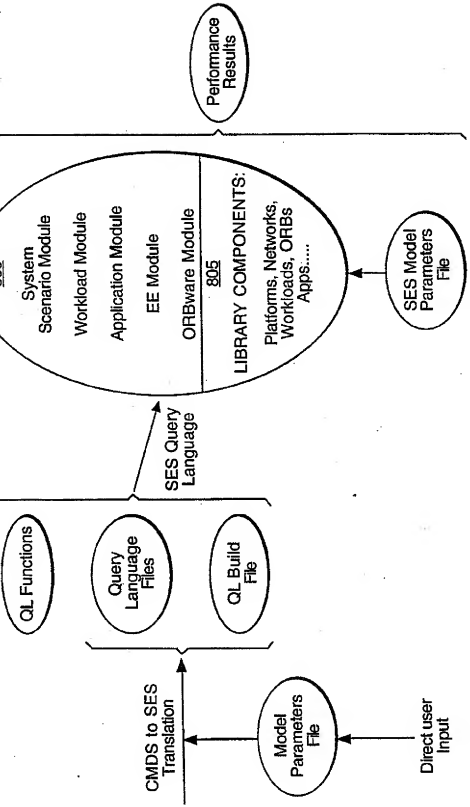

Performance Results
SES/WB
800
System Scenario Module
Workload Module
Application Module
EE Module
ORBware Module
805
LIBRARY COMPONENTS:
Platforms, Networks, Workloads, ORBs Apps.....
SES Model Parameters File
SES Query Language
QL Functions
Query Language Files
QL Build File
CMDS to SES Translation
Model Parameters File
Direct user Input

Fig.8.

Fig.9.

Cache example: Application-Class Diagram

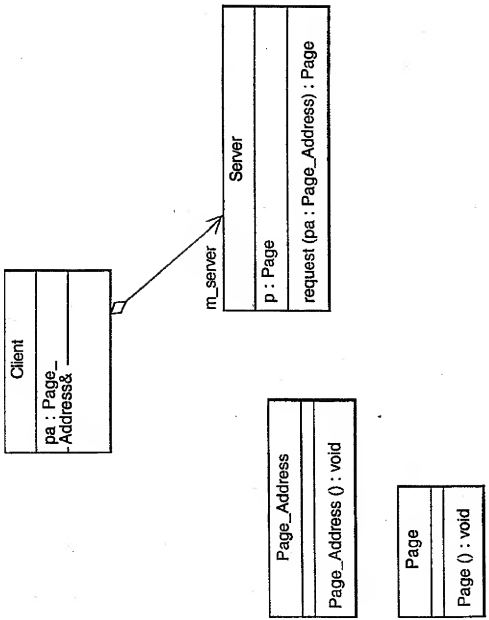


Fig.10.

Cache example: Application-Interaction Diagram

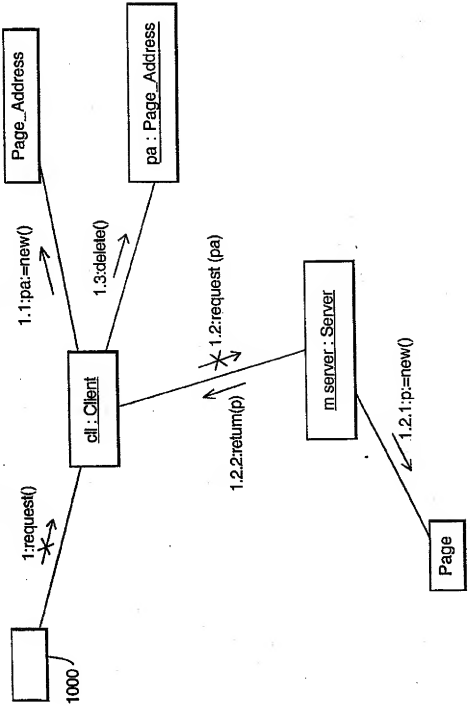


Fig.11.

Cache example: Execution Environment - Class Diagram

Fig.12.

Cache example: Workload - Class Diagram

Fig.13.
Cache example: System Scenario Diagram

Fig.14.

Model Parameters file

```
warm_up_time      0
run_time      100

Application Parameters
Page_Address.collect_stats      0
Server.collect_stats      1
Client.collect_stats      1
Page.collect_stats      0

EOF
```

Fig.15.

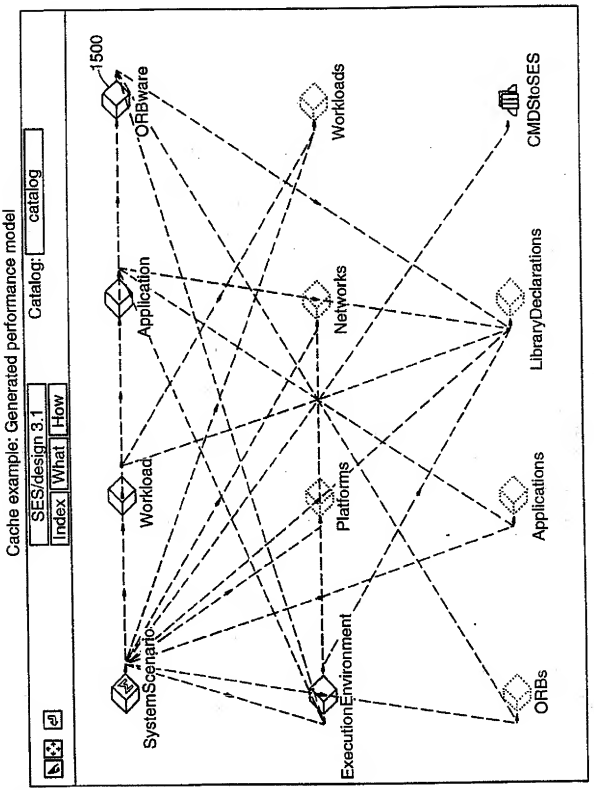
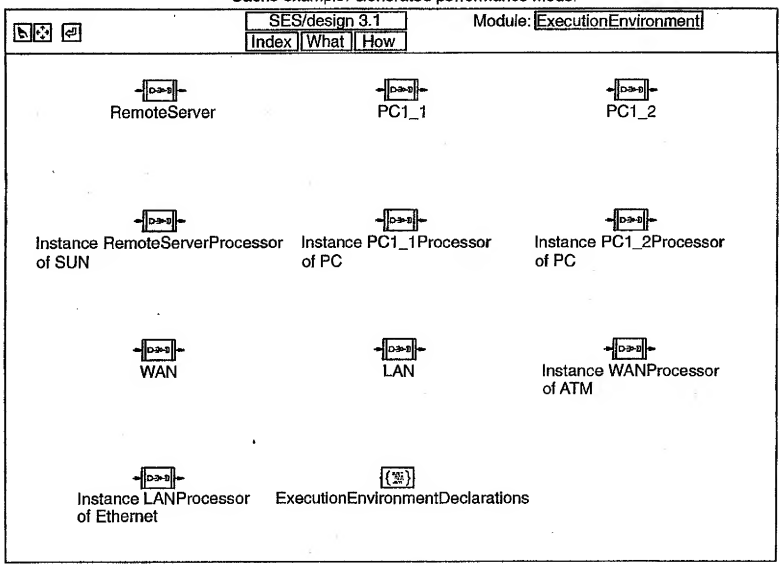

Cache example: Generated performance model
SES/design 3.1
Index
What
How
Catalog:
catalog
1500
ORBware
Application
Workload
Workloads
Networks
Platforms
SystemScenario
ExecutionEnvironment
ORBs
Applications
LibraryDeclarations
CMDStoSES

Fig.16.



SUBSTITUTE SHEET (RULE 26)

Fig.17.

Cache example: Generated performance model
SES/design 3.1
Module: Workload
Index
What
How
Instance WL1_1
of Closed_Thinker
Instance WL1_2
of Closed_Thinker

Fig.18.

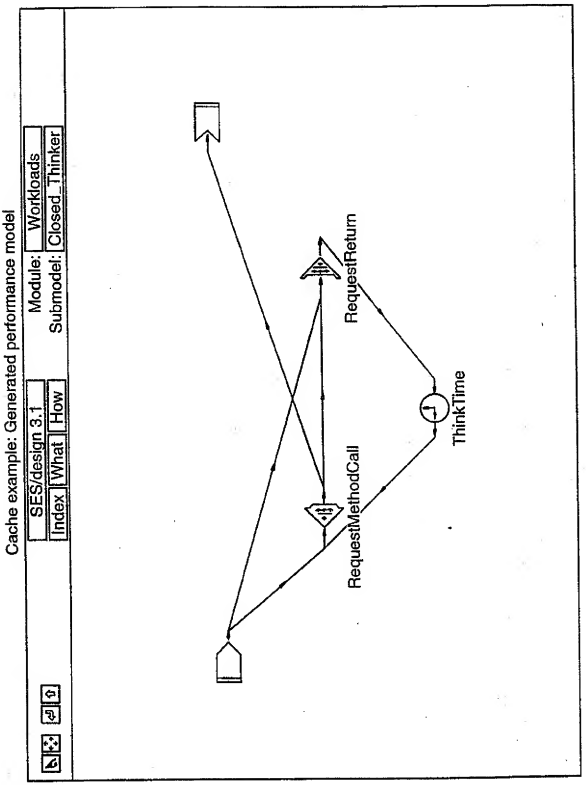

Cache example: Generated performance model
SES/design 3.1
Index
What
How
Module: Workloads
Submodel: Closed_Thinker
RequestMethodCall
RequestReturn
ThinkTime

Fig.19.

Cache example: Generated performance model
SES/design 3.1
Module: Application
Index
What
How
Page
Page_Address
Server
Client

Fig.20.

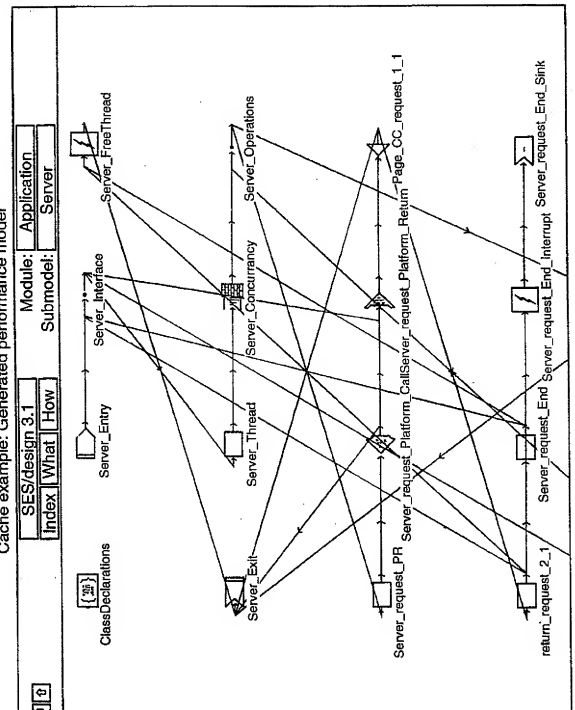

Cache example: Generated performance model
SES/design 3.1
Index | What | How
Module: Application
Submodel: Server
ClassDeclarations
Server_Entry
Server_Interface
Server_FreeThread
Server_Exit
Server_Thread
Server_Concurrency
Server_Operations
Server_request_PR
Server_request_Platform_CallServer_request_Platform_Return
Page_CC_request_1_1
return_request_2_1
Server_request_End
Server_request_End_Interrupt
Server_request_End_Sink

Fig.21.

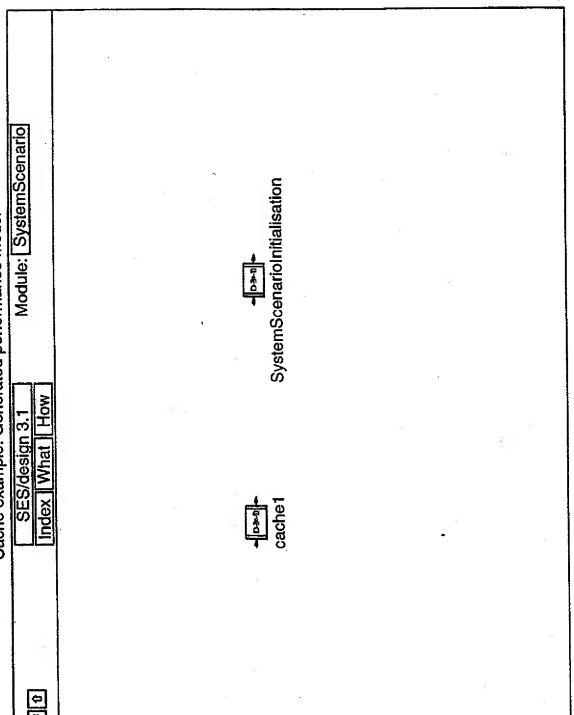

Cache example: Generated performance model
SES/design 3.1
Module: SystemScenario
Index
What
How
cache1
SystemScenarioInitialisation

Fig.22.

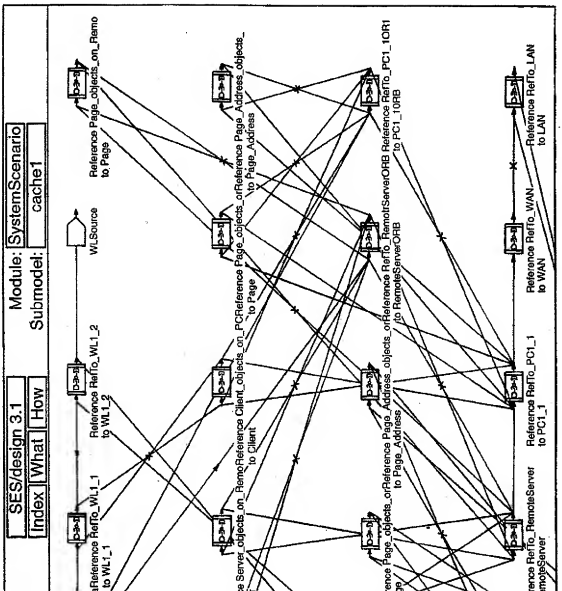
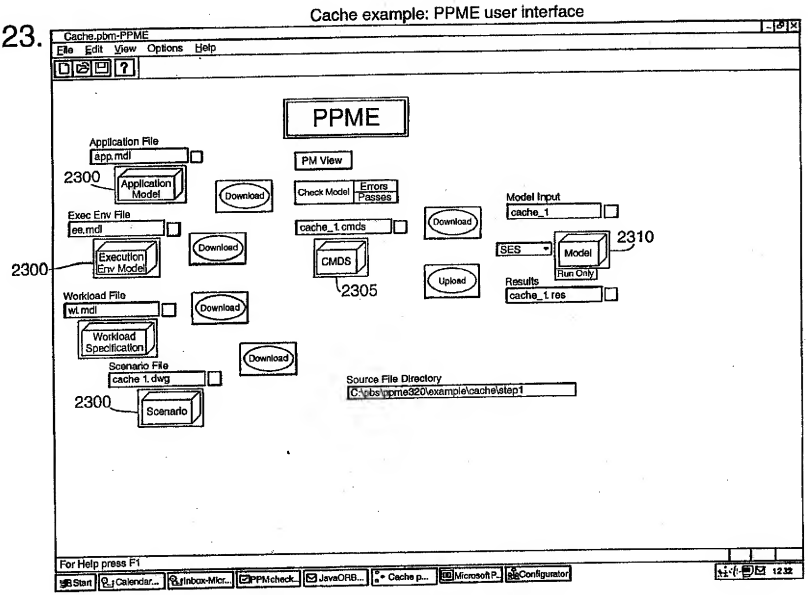

Cache example: Generated performance model
SES/design 3.1
Index | What | How
Module: SystemScenario
Submodel: cache1
Reference RefTo_SystemScenarioInitialisation SystemScenarioInitialisation
Reference RefTo_WI1_1 to WI1_1
Reference RefTo_WI1_2 to WI1_2
WLSource
Reference Page_objects_on_Remo to Page
Reference Page_Address_objects_ to Page_Address
Reference Server_objects_on_Remo to Server
Reference Client_objects_on_PC to Client
Reference Page_objects_on_PC to Page
Reference Page_Address_objects_on_PC to Page_Address
Reference RefTo_RemoteServer to RemoteServer
Reference RefTo_PC1_1 to PC1_1
Reference RefTo_WAN to WAN
Reference RefTo_LAN to LAN
Reference RefTo_PC1_20RB to PC1_20RB
Reference RefTo_RemoteServerORB to RemoteServerORB
Reference RefTo_PC1_10R1 to PC1_10RB

Fig.23.



SUBSTITUTE SHEET (RULE 26)

Fig.24.

Cache example:CMDS
CMDSbrowser-cache_1.cmds
Help Edit View Window Help
cache_1.cmds
System
Instances
System_Scenario-00000
Instances
App_Object-00000
Platform_Instance-00001
Platform_Instance-00002
Network_Instance-00003
Network_Instance-00004
App_Object-00005
Workload_Instance-00006
Workload_Instance-00007
App_Object-00008
Platform_Instance-00009
Slot-connections
Connection-LAN->LAN
Connection-LAN->WAN
Connection-WAN->LAN
Connection-WAN->WAN
List-routes
Route-00000
Route-00001
Route-00002
Route-00003
Route-00004
Route-00005
Package-types
Instances
Package-00000
List-contents
Package-00000
List-contents
Package-00000
List-attributes
Set-behaviour
List-dependancies
List-operations
List-properties
List-relationships
Views
00000
System_Scenario

Fig.25.

Top Level

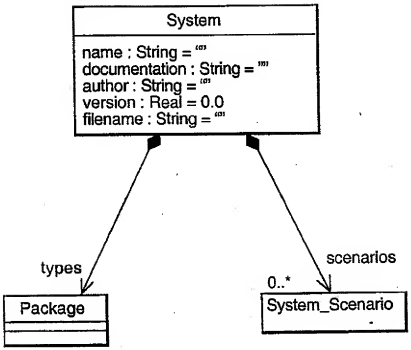
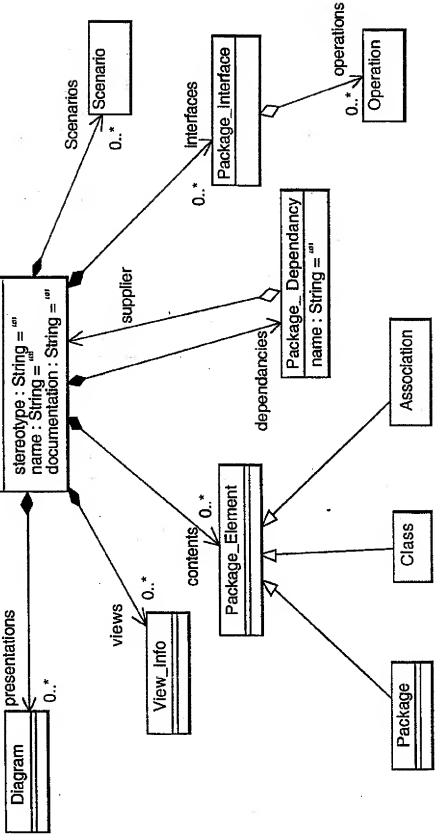


Fig.26


Package
Package
stereotype : String = ""
name : String = ""
documentation : String = ""
Scenarios
0..*
Scenario
interfaces
0..*
Package_Interface
operations
0..*
Operation
supplier
dependancies
Package_ Dependancy
name : String = ""
presentations
0..*
Diagram
views
0..*
View_Info
contents
0..*
Package_Element
Association
Class
Package

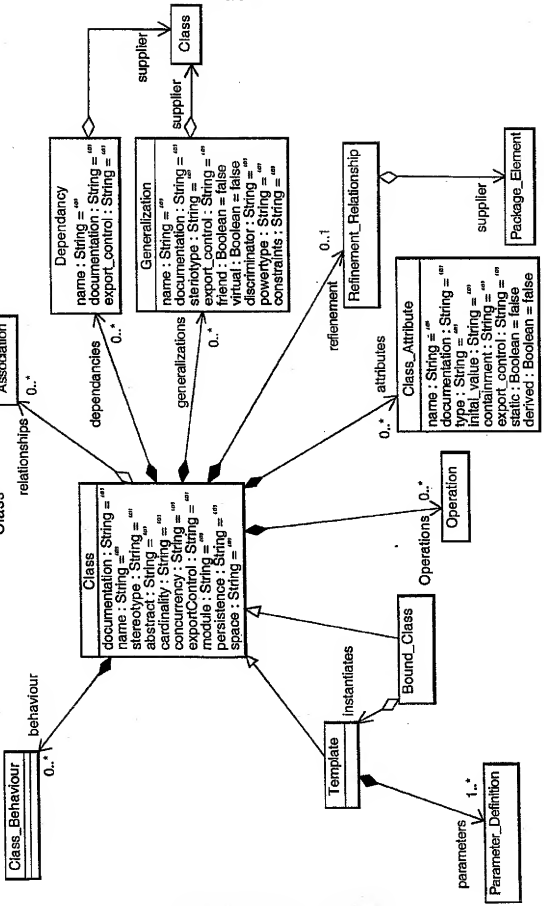

Class
Class_Behaviour
behaviour
0..*
Class
documentation : String = ""
name : String = ""
stereotype : String = ""
abstract : String = "no"
cardinality : String = "1"
concurrency : String = "sequential"
exportControl : String = "public"
module : String = ""
persistence : String = "transient"
space : String = ""
relationships
Assodation
0..*
dependancies
Dependancy
name : String = ""
documentation : String = ""
export_control : String = "public"
supplier
Class
generalizations
Generalization
name : String = ""
documentation : String = ""
stereotype : String = ""
export_control : String = "public"
friend : Boolean = false
virtual : Boolean = false
discriminator : String = ""
powertype : String = ""
constraints : String = ""
refinement
0..1
Refinement_Relationship
supplier
Package_Element
attributes
Class_Attribute
name : String = ""
documentation : String = ""
type : String = ""
Inital_value : String = ""
containment : String = "by_value"
export_control : String = "public"
static : Boolean = false
derived : Boolean = false
Operations
Operation
Template
Instantiates
Bound_Class
parameters
1..*
Parameter_Definition

Fig.27.

Fig.28.

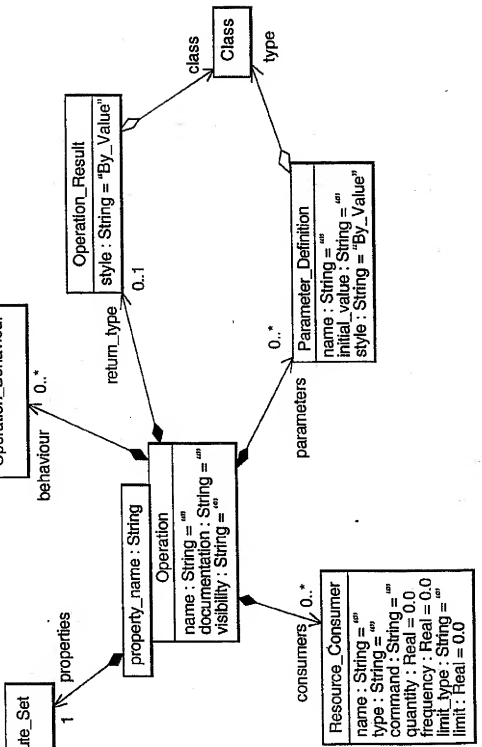

Attribute_Set
1
properties
property_name : String
Operation
name : String = ""
documentation : String = ""
visibility : String = ""
Operation_Behaviour
behaviour
0..*
return_type
0..1
Operation_Result
style : String = "By_Value"
class
Class
type
0..*
parameters
Parameter_Definition
name : String = ""
initial_value : String = ""
style : String = "By_Value"
consumers
0..*
Resource_Consumer
name : String = ""
type : String = ""
command : String = ""
quantity : Real = 0.0
frequency : Real = 0.0
limit_type : String = ""
limit : Real = 0.0

Fig.29.

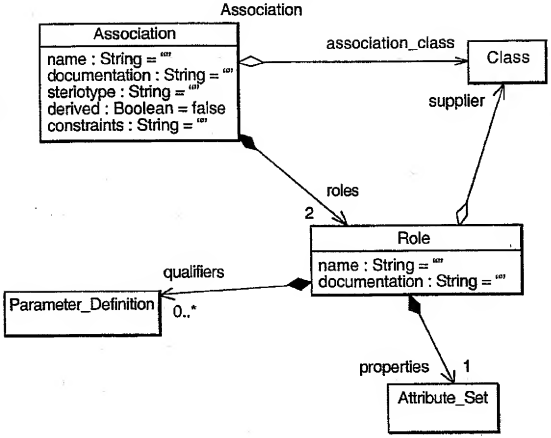

Association
Association
name : String = ""
documentation : String = ""
steriotype : String = ""
derived : Boolean = false
constraints : String = ""
association_class
Class
supplier
roles
2
Role
name : String = ""
documentation : String = ""
qualifiers
Parameter_Definition
0..*
properties
1
Attribute_Set

Fig.30.

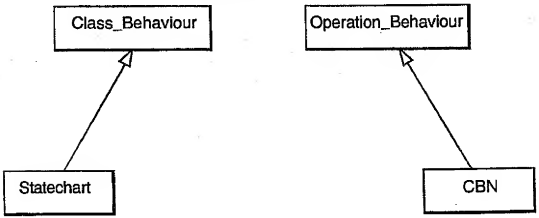

Behaviours
Class_Behaviour
Operation_Behaviour
Statechart
CBN

Fig.31.

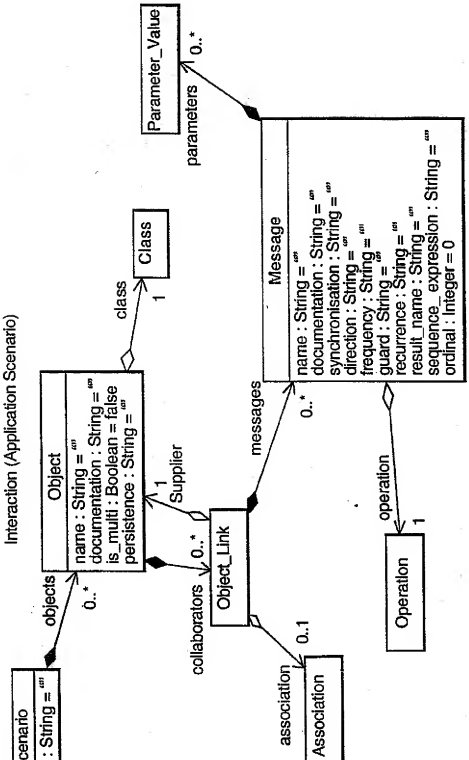
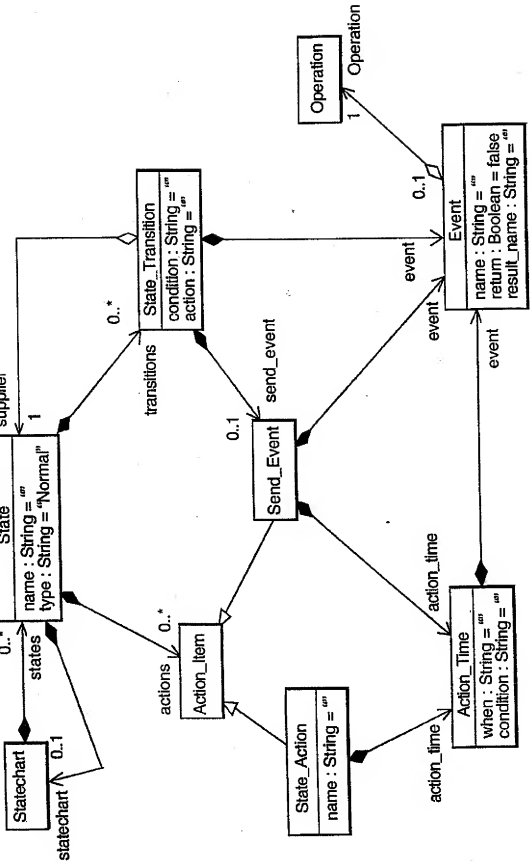

Interaction (Application Scenario)
Scenario
name : String = ""
objects
0..*
Object
name : String = ""
documentation : String = ""
is_multi : Boolean = false
persistence : String = ""
class
1
Class
Parameter_Value
0..*
parameters
1
Supplier
0..*
collaborators
Object_Link
messages
0..*
Message
name : String = ""
documentation : String = ""
synchronisation : String = ""
direction : String = ""
frequency : String = ""
guard : String = ""
recurrence : String = ""
result_name : String = ""
sequence_expression : String = ""
ordinal : Integer = 0
association
0..1
Association
operation
1
Operation

Fig.32.


Statechart
statechart
0..1
0..*
states
State
name : String = ""
type : String = "Normal"
supplier
1
0..*
transitions
State_Transition
condition : String = ""
action : String = ""
0..*
actions
Action_Item
0..1
send_event
Send_Event
State_Action
name : String = ""
action_time
Action_Time
when : String = ""
condition : String = ""
action_time
event
Event
name : String = ""
return : Boolean = false
result_name : String = ""
0..1
event
event
Operation
1
Operation

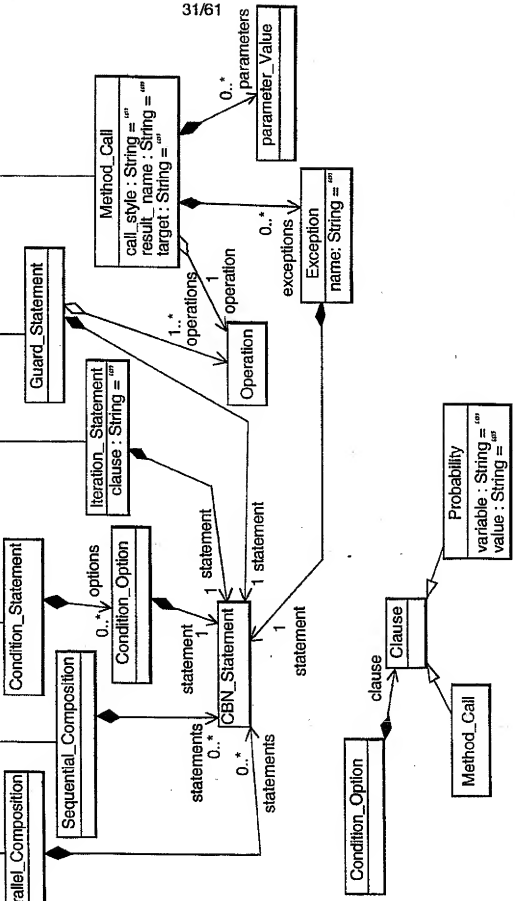

CBN_Statement
Condition_Statement
Sequential_Composition
Parallel_Composition
Condition_Option
Iteration_Statement
clause : String = ""
Guard_Statement
Method_Call
call_style : String = ""
result_name : String = ""
target : String = ""
parameters
parameter_Value
Operation
operations
operation
exceptions
Exception
name: String = ""
options
statement
statements
clause
Clause
Probability
variable : String = ""
value : String = ""
Method_Call

Fig.33.
CBN

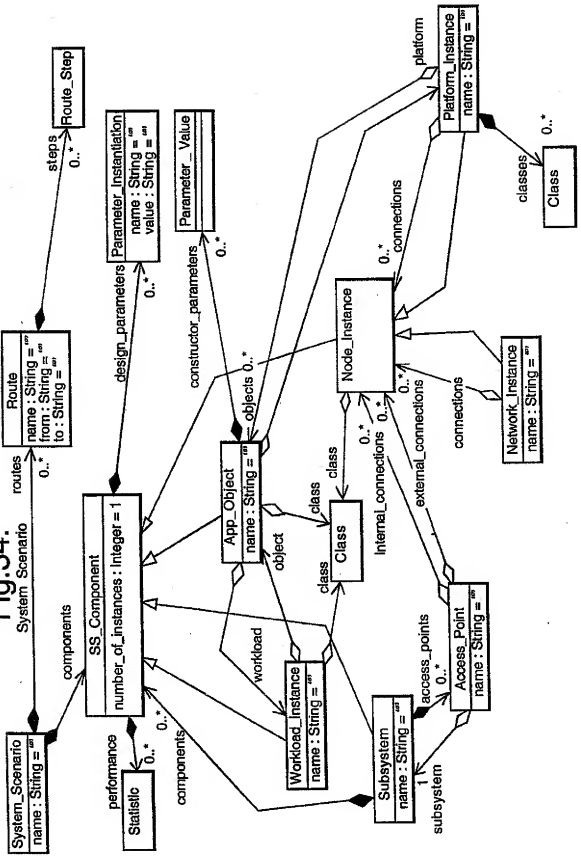

System_Scenario
name : String = ""
System_Scenario
components
SS_Component
number_of_instances : Integer = 1
routes
Route
name : String = ""
from : String = ""
to : String = ""
steps
Route_Step
design_parameters
Parameter_Instantiation
name : String = ""
value : String = ""
constructor_parameters
Parameter_Value
objects
App_Object
name : String = ""
class
Class
Internal_connections
external_connections
Node_Instance
connections
platform
Platform_Instance
name : String = ""
classes
Class
Network_Instance
name : String = ""
connections
performance
StatisItc
components
workload
Workload_Instance
name : String = ""
Subsystem
name : String = ""
subsystem
access_points
Access_Point
name : String = ""

Fig.34.

Fig.35.

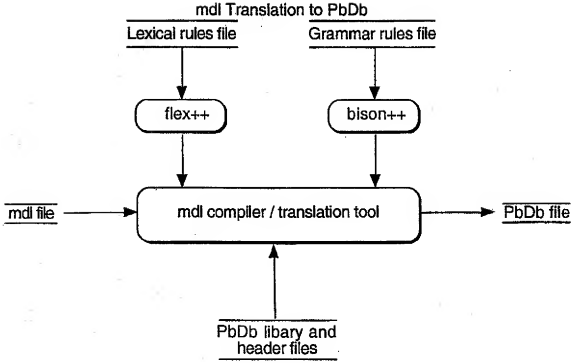

mdl Translation to PbDb
Lexical rules file
Grammar rules file
flex++
bison++
mdl file
mdl compiler / translation tool
PbDb file
PbDb libary and header files

Fig.36.

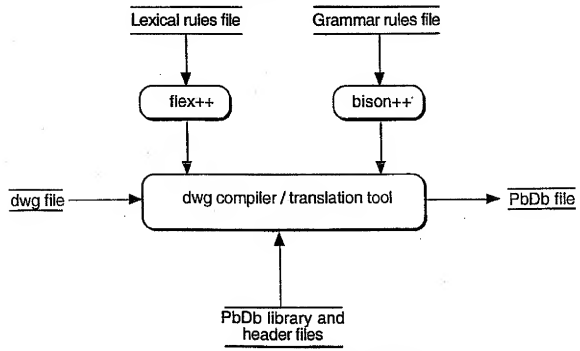

dwg translation to PbDb
Lexical rules file
Grammar rules file
flex++
bison++
dwg file
dwg compiler / translation tool
PbDb file
PbDb library and header files

Fig.37.

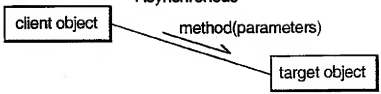


Fig.38.

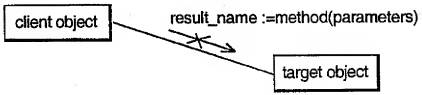


Fig.39.

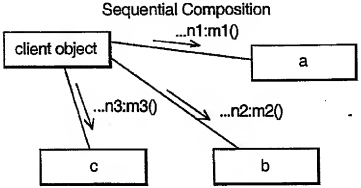


Fig.40.

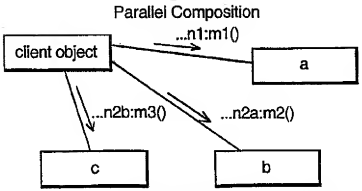


Fig.41.

Iteration Statement

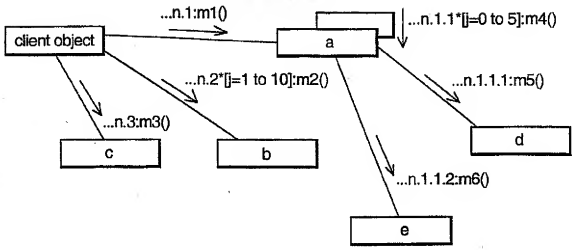


Fig.42.

Conditional Statement

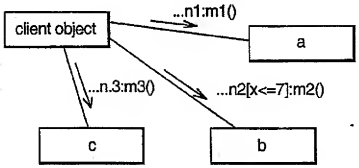


Fig.43.

Guard Statement

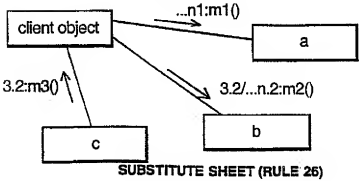


Fig.44.

Probabilistic Conditional Statement

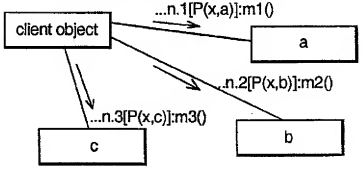


Fig.45.

Video Machine Statechart

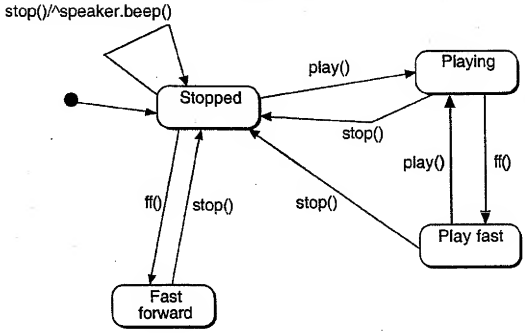


Fig.46.

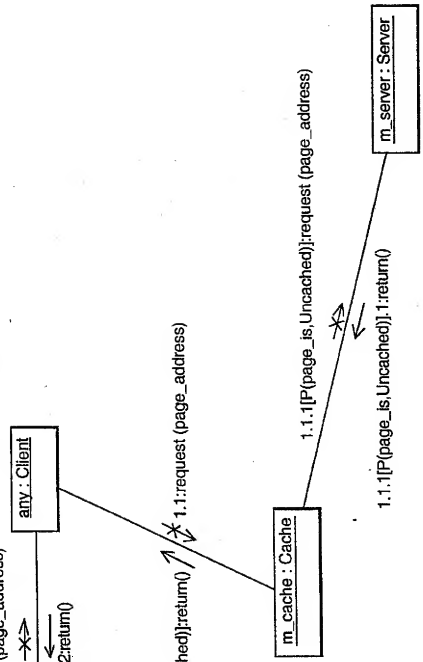

Uncached Page
1:request (page_address)
any : Client
fred : Browsing User
1.2:return()
1.1.2[P(page_is,Uncached)]:return()
1.1:request (page_address)
m_cache : Cache
1.1.1[P(page_is,Uncached)]:request (page_address)
1.1.1[P(page_is,Uncached)].1:return()
m_server : Server

Fig.47.

Freshly Cached Page

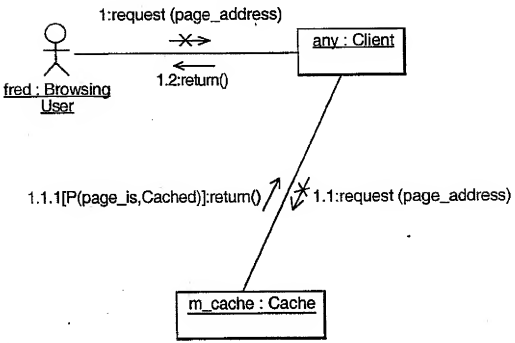


Fig.48.

Suspect Valid Page

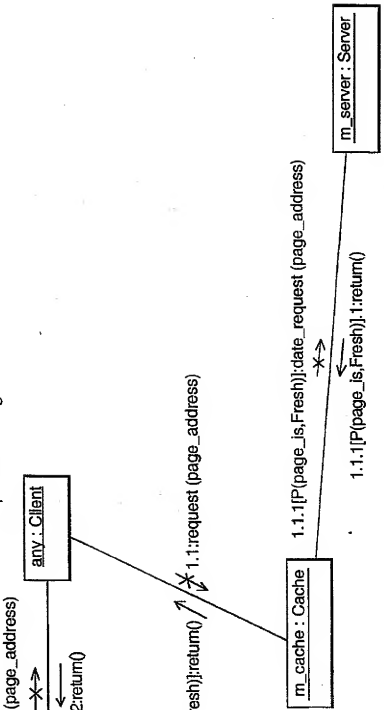


Fig.49.

Suspect Invalid Page

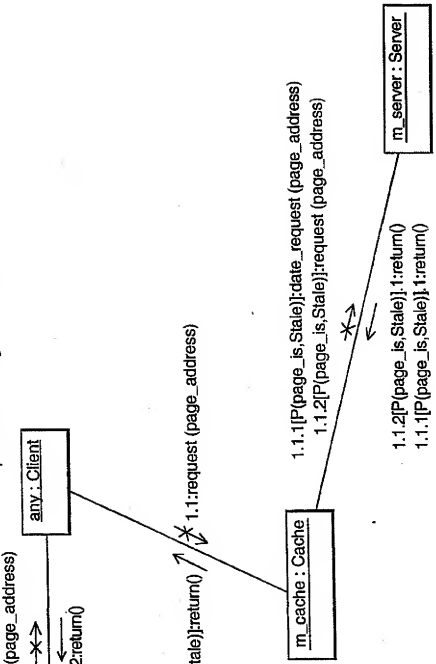


Fig.50.

CMDS to SES translation process
CMDS
CMDS Data Base
Query Language File
SES Performance Model

Fig.51.

Application and Execution Environment Object Interaction
a:AA
process(data_size)
pcl:PC

Fig.52.

Meta Model Representation of Application Interaction
pca
pcb
NW5
NW6
NW3
Source Model
a:AA
f(x)
b:BB
Translation Process
Meta Model
a:AA
b:BB
Application Layer
f(x)
f(x)
pcaOA:OA
pcbOA:OA
ORB Layer
transmit(f,x)
transmit(f,x)
transmit(f,x)
pca:PC
pcb:PC
Execution Environment Layer

Fig.53.
Network Graph

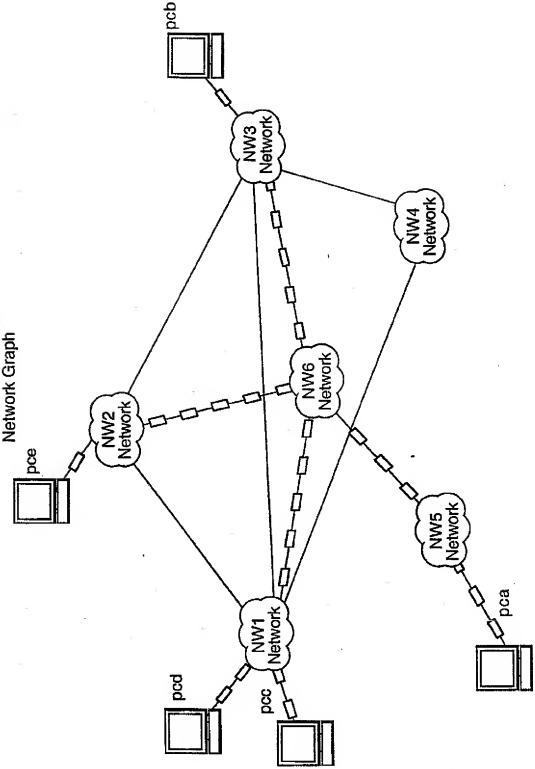


Fig.54.

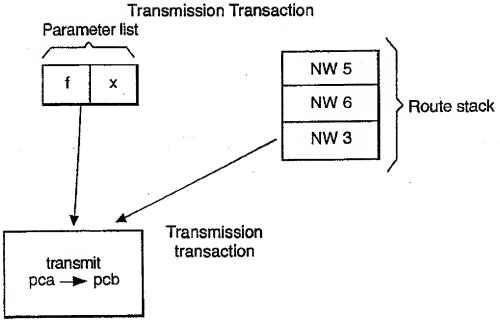

Transmission Transaction
Parameter list
f
x
NW 5
NW 6
NW 3
Route stack
Transmission transaction
transmit
pca → pcb

Fig.55.

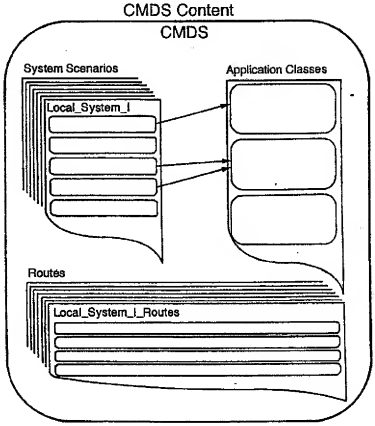

CMDS Content
CMDS
System Scenarios
Application Classes
Local_System_I
Routes
Local_System_I_Routes

Fig.56.

SES/design 3.1
Index
What
How
Catalog:
catalog
SystemScenario
Workload
Application
ORBware
ExecutionEnvironment
Platforms
Networks
Workloads
ORBs
Applications
LibraryDeclarations
CMDStoSES
CMDStoSESLibrary

Fig.57.

Configurator System Scenario for Caching Example

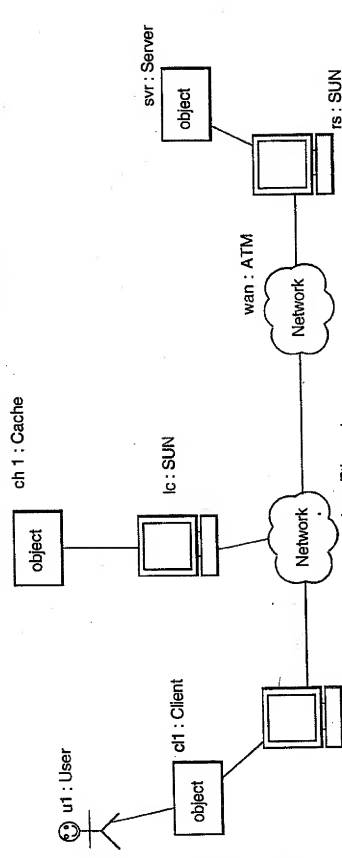
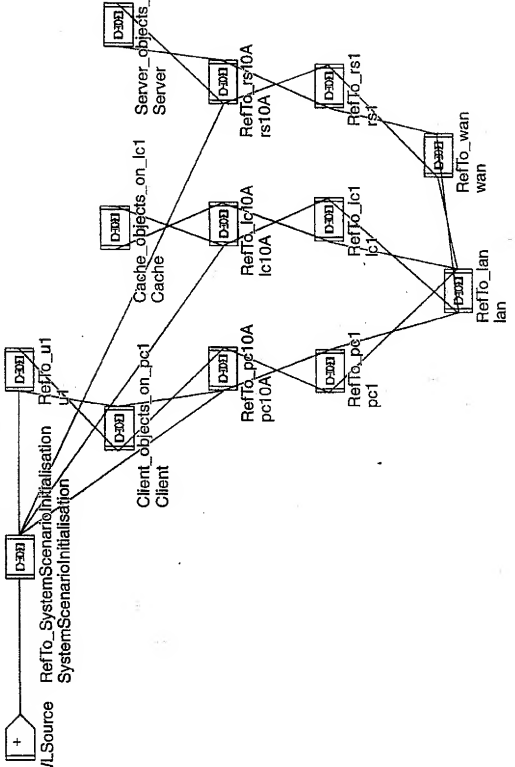


Fig.58.

SES system scenario for Caching example



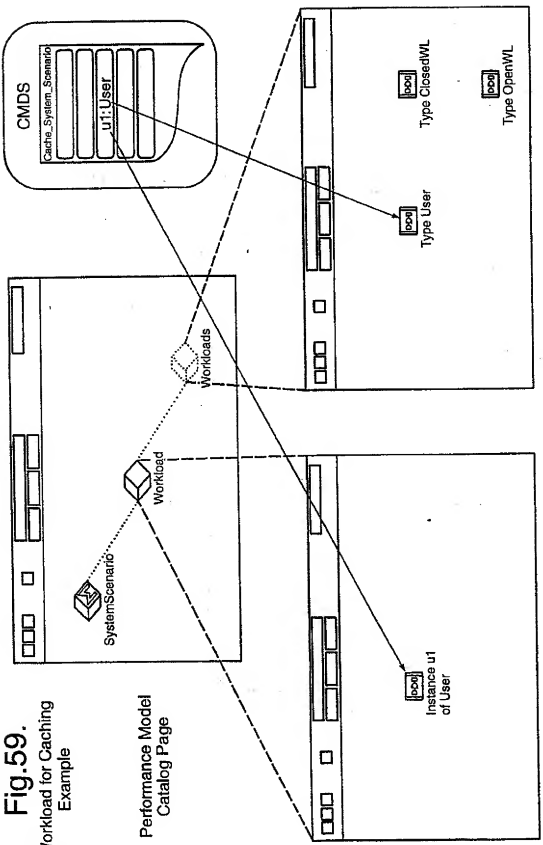


Fig.59.
Workload for Caching Example

Performance Model Catalog Page

Fig.60.
ORB layer for Caching Example

Performance Model Catalog Page

Fig.61.

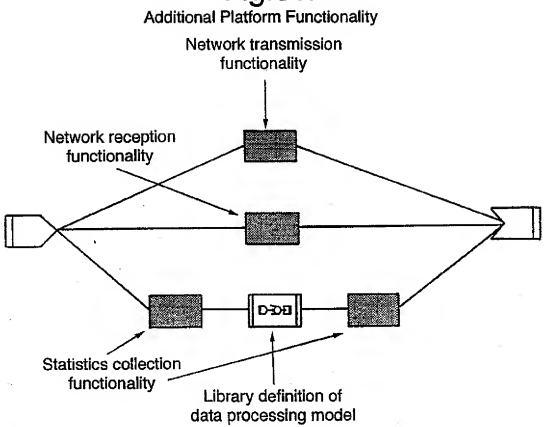
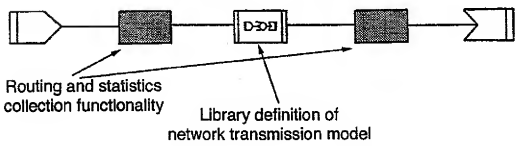

Additional Platform Functionality
Network transmission functionality
Network reception functionality
Statistics collection functionality
Library definition of data processing model

Fig.62.


Additional Network Functionality
Routing and statistics collection functionality
Library definition of network transmission model

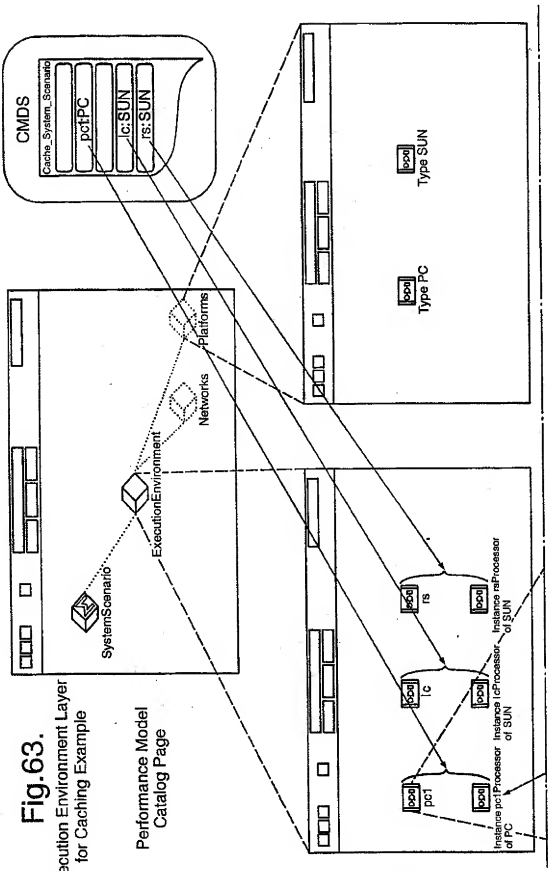


Fig.63.
Execution Environment Layer for Caching Example

Performance Model Catalog Page

Fig.63 (Cont).

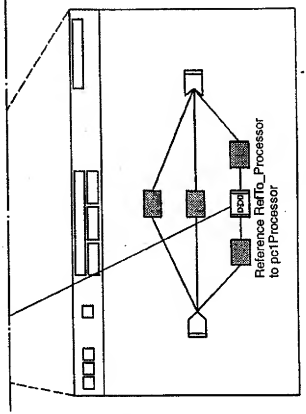


Fig.64.

Concurrency Control for Application Layer Objects
Single blocking node in behaviour model for class A
Object free
Object busy
T2 T5 T1 T4 T7 T6 T3 T8
Order in which method call transaction arrive at behaviour model for class A
Table of queues for each object of class A
A
a1
T2 T4 T8
a2
T1 T6
a3
T5 T7 T3
Order in which method calls are made on each object

Fig.65.

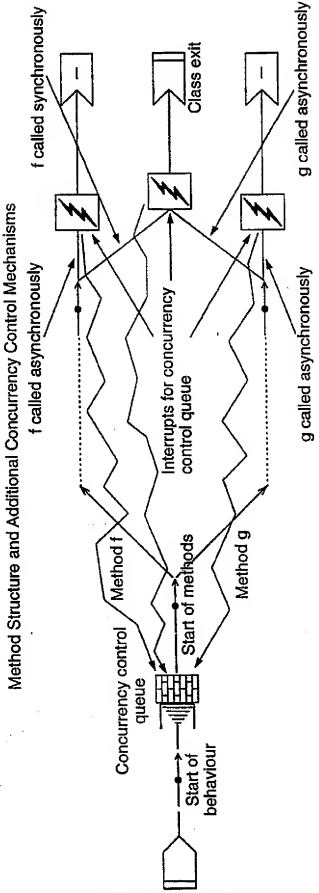

Method Structure and Additional Concurrency Control Mechanisms
f called asynchronously
f called synchronously
Interrupts for concurrency control queue
Class exit
g called asynchronously
g called asynchronously
Method f
Method g
Start of methods
Concurrency control queue
Start of behaviour

Fig.66.

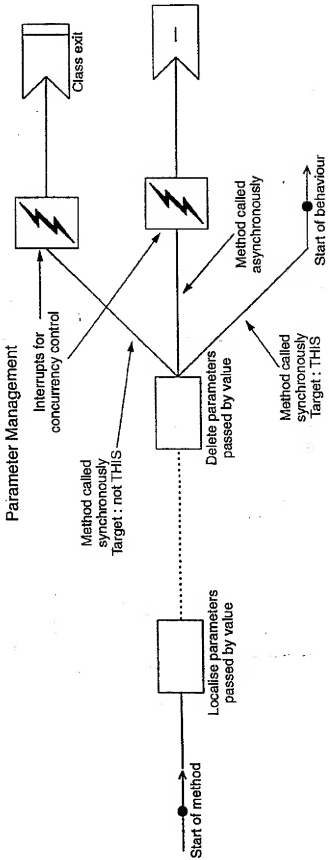

Parameter Management
Start of method
Localise parameters passed by value
Method called synchronously Target : not THIS
Interrupts for concurrency control
Delete parameters passed by value
Method called asynchronously
Method called synchronously Target : THIS
Start of behaviour
Class exit

Fig.67.

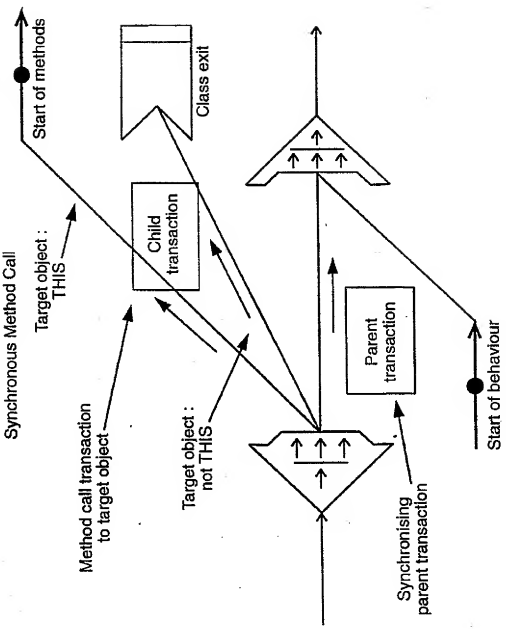

Synchronous Method Call
Start of methods
Target object : THIS
Class exit
Child transaction
Method call transaction to target object
Target object : not THIS
Parent transaction
Synchronising parent transaction
Start of behaviour

Fig.68.

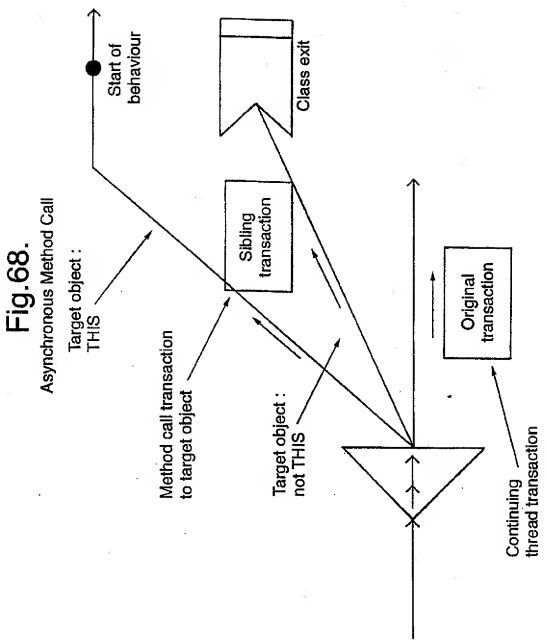

Asynchronous Method Call
Target object : THIS
Start of behaviour
Method call transaction to target object
Sibling transaction
Class exit
Target object : not THIS
Original transaction
Continuing thread transaction

Fig.69.

Return Method Call

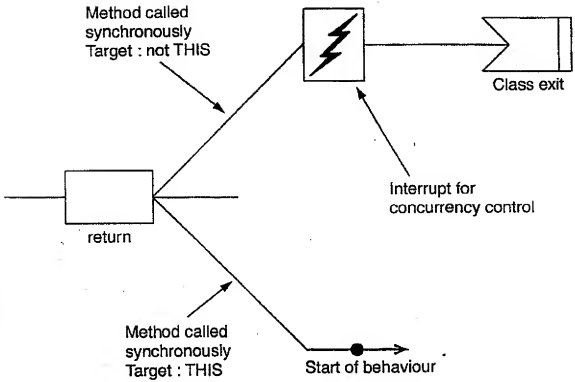


Fig.70.

Conditional Statement

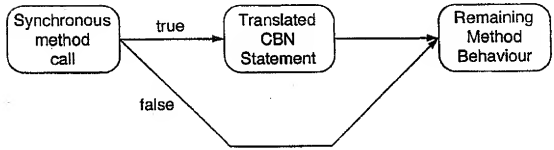


Fig.71.

Probabilistic Condition Statement

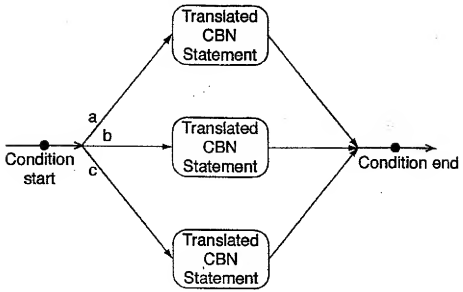


Fig.72.

Sequential Statement

Fig.73.

Guard Statement

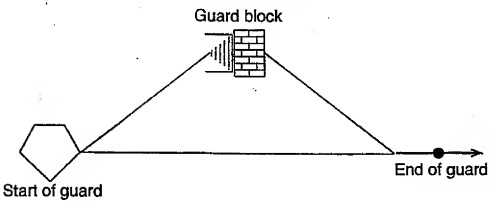


Fig.74.

Transmission Transaction

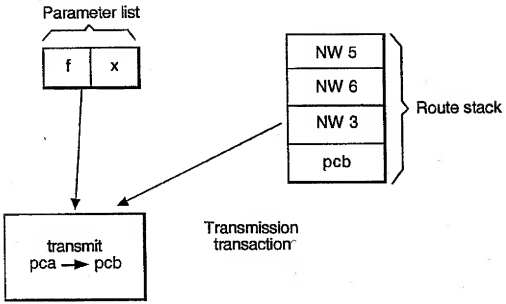


Fig.75.

SES/design 3.1
Index
What
How
Module: ORBs
Submodel: DefaultORB
ORBDeclarations
UpDateTables
DeleteDestructor
Delete Object
ToRemoteFromLocal
Supplier Remote
Supplier Local
Caller Local
New Object
UpdateORBTable
ToLocalFromLocal
Caller Remote
ToLocalFromRemote
Supplier Local
Wrong ORB
WrongORB